Verifying constant-time implementations by abstract interpretation

Sandrine Blazy, David Pichardie, Alix Trieu

Verifying Constant-Time Implementations by Abstract Interpretation

Sandrine Blazy [a], David Pichardie [a] and Alix Trieu [a]
[a] *Univ Rennes, Inria, CNRS, IRISA, France*

Abstract. Constant-time programming is an established discipline to secure programs against timing attackers. Several real-world secure C libraries such as NaCl, mbedTLS, or Open Quantum Safe, follow this discipline. We propose an advanced static analysis, based on state-of-the-art techniques from abstract interpretation, to report time leakage during programming. To that purpose, we analyze source C programs and use full context-sensitive and arithmetic-aware alias analyses to track the tainted flows.

We give semantic evidence of the correctness of our approach on a core language. We also present a prototype implementation for C programs that is based on the CompCert compiler toolchain and its companion Verasco static analyzer. We present verification results on various real-world constant-time programs and report on a successful verification of a challenging SHA-256 implementation that was out of scope of previous tool-assisted approaches.

Keywords: abstract interpretation, constant-time programming, timing attacks, verification of C implementations

1. Introduction

To protect their implementations, cryptographers follow a very strict programming discipline called constant-time programming. They avoid branchings controlled by secret data as an attacker could use timing attacks, which are a broad class of side-channel attacks that measure different execution times of a program in order to infer some of its secret values [1–4]. They also avoid memory loads and stores indexed by secret data because of cache-timing attacks. Several real-world secure C libraries such as NaCl [5], mbedTLS [6], or Open Quantum Safe [7], follow this programming discipline.

Despite the name, constant-time programming does not ensure that the program runs in constant-time, only that its running time does not depend on secrets. For instance, two different branches may have different execution times, but it is not harmful if the branching cannot leak information on secrets. Balancing the running time of branches by adding no-op instructions may seem an attractive solution, but they may be removed by compilers, and it is still open to other attacks as illustrated by [8].

The constant-time programming discipline requires the transformation of source programs. These transformations may be tricky and error-prone, mainly because they involve low-level features of C and non-standard operations (e.g., bit-level manipulations). We argue that programmers need tool assistance to use this programming discipline. First, they need feedback at the source level during programming, in order to verify that their implementation is constant time and also to understand why a given implementation is not constant time as expected. Second, they need to trust that their compiler will not break the security of source programs when translating the guarantees obtained at the source level. Indeed, compiler optimizations could interfere with the previous constant-time transformations performed by the programmer. In this paper, we choose to implement a static analysis at source level to simplify error

reporting, but couple the static analyzer to the highly trustworthy CompCert compiler [9]. This strategic design choice allows us to take advantage of static analysis techniques that would be hard to apply at the lowest program representation levels.

Static analysis is frequently used for identifying security vulnerabilities in software, for instance to detect security violations pertaining to information flow [10–12]. In this paper, we propose an advanced static analysis, based on state-of-the-art techniques from abstract interpretation [13] (mainly fixpoint iterations operating over source programs, use of widening operators, computations performed by several abstract domains including a memory abstract domain handling pointer arithmetic), to report time leakage during programming.

Data originating from a statement where information may leak is tainted with the lowest security level. Our static analysis uses two security levels, that we call secret (high level) and public (low level); it analyzes source C programs and uses full context-sensitive (i.e., the static analysis distinguishes the different invocations of a same function) and arithmetic-aware alias analyses (i.e., the cells of an array are individually analyzed, even if they are accessed using pointer dereferencing and pointer arithmetic) to track the tainted flows.

We follow the abstract interpretation methodology: we design an abstract interpreter that executes over security properties instead of concrete values, and use approximation of program executions to perform fixpoint computations. We hence leverage the inference capabilities of advanced abstract interpretation techniques as relational numeric abstractions [14], abstract domain collaborations [15], arithmetic-aware alias analysis [16, 17], to build a very precise taint analysis on C programs. As a consequence, even if a program uses the same memory block to store both secret and public values during computations, our analysis should be able to track it, without generating too many spurious false alarms. This programming pattern appears in real-world implementations, such as the SHA-256 implementation in NaCl that we are able to analyze.

In this paper, we make the following contributions:

- We define a new methodology for verifying constant-time security of C programs. Our static analysis is fully automatic and sound by construction.
- We instrument our approach in the Verasco static analyzer [18]. Verasco is a formally-verified static analyzer, that is connected to the formally-verified CompCert C compiler. We thus benefit from the CompCert correctness theorem, stating roughly that a compiled program behaves as prescribed by the semantics of its source program.
- We report our results obtained from a benchmark of representative cryptographic programs that are known to be constant time. Thanks to the precision of our static analyzer, we are able to analyze programs that are out of reach of state-of-the-art tools.

This paper is organized as follows. First, Section 2 presents the Verasco static analyzer. Then, Section 3 explains our methodology and details our abstract interpreter. Section 4 describes the experimental evaluation of our static analyzer. Related work is described in Section 5, followed by conclusions.

2. The Verasco Abstract Interpreter

Verasco is a static analyzer based on abstract interpretation that is formally verified in Coq [18]. Its proof of correctness ensures the absence of runtime errors (such as out-of-bound array accesses, null pointer dereferences, and arithmetic exceptions) in the analyzed C programs. Verasco relies on several

abstract domains, including a memory domain that finely tracks properties related to memory contents, taking into account type conversions and pointer arithmetic [17].

Verasco is connected to the CompCert formally-verified C compiler, that is also formally verified in Coq [9]. Its correctness theorem is a semantics preservation theorem; it states that the compilation does not introduce bugs in compiled programs. More precisely, Verasco operates over C#minor a C-like language that is the second intermediate language in the CompCert compilation pipeline.

Verasco raises an alarm as soon as it detects a potential runtime error. Its correctness theorem states that if Verasco returns no alarm, then the analyzed program is *safe* (i.e., none of its observable behaviors is an undefined behavior, according to the C#minor semantics). The design of Verasco is inspired by Astrée [19], a milestone analyzer that was able to successfully analyze realistic safety-critical software systems for aviation and space flights. Verasco follows a similar modular architecture as Astrée, that is shown in Figure 1.

First, at the bottom of the figure, a large hub of numerical abstract domains is provided to infer numerical invariants on programs. These properties can be *relational* as for example $j + 1 \leqslant i \leqslant j + 2$ in a loop (with *Octagons* or *Polyhedra* abstract domains). All these domains finely analyze the behavior of machine integers and floating-points (with potential overflows) while unsound analyzers would assume ideal arithmetic. They are connected all-together via *communication channels* that allow each domain to improve its own precision via specific queries to other domains. As a consequence, Verasco is able to infer subtle numerical invariants that require complex reasoning about linear arithmetic, congruence and symbolic equalities.

Second, on top of these numerical abstractions sits an abstract memory functor [17] that tracks fine-grained aliases and interacts with the numerical domains. This functor can choose to represent every cell of the same memory block with a single property, or to finely track each specific property of every position in the block. Contrary to many other alias analyses, this approach allows us to reason on local and global variables with the same level of precision, even when the memory addresses are manipulated by the programmer. Some unavoidable approximations are performed when the target of a memory dereference corresponds to several possible targets, but Verasco makes the impact of such imprecision as limited as possible. Because of ubiquitous pointer arithmetic in C programs (even simple array accesses are represented via pointer arithmetic in C semantics), the functor needs to ask advanced symbolic numerical queries to the abstract numerical domain below it. In return, its role is to hide from them the load and store operations, and only communicate via symbolic numerical variables.

Third, the last piece of the analyzer is an advanced abstract interpreter that builds a fixpoint for the analysis result. This task is a bit more complex than in standard dataflow analysis techniques that look for the least solution of dataflow equation systems. In such settings, each equation is defined by means of monotone operators in a well-chosen lattice without infinite ascending chains. By computing the successive iterates of the transfer functions attached to each equation, starting from a bottom element, the fixpoint computation always terminates on the least element of the lattice that satisfies all equations. In contrast, the Verasco abstract interpreter relies on infinite lattices, where widening and narrowing operators [13] are used for ensuring and accelerating the convergence. Smart iteration strategies are crucial when using such accelerating operators because they directly impact the precision of the analysis diagnosis. Verasco builds its strategy by following the structure of the program. On every program loop, it builds a local fixpoint using accelerating techniques. At every function call, it makes a recursive call of the abstract interpreter on the body of the callee. The callee may be resolved thanks to the state abstraction functor in presence of function pointers. The recursive nature of the abstract interpreter makes the analysis

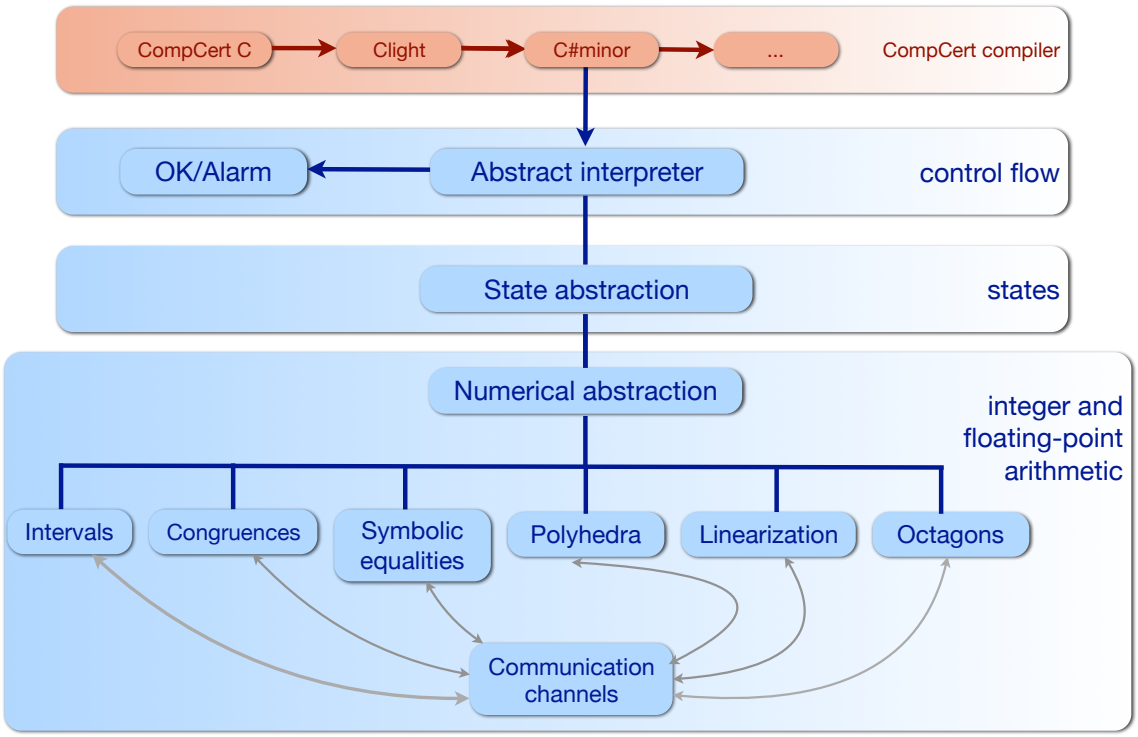

Figure 1. Architecture of the Verasco static analyzer

very precise because each function is independently analyzed as many times as there are calling contexts that invoke it.

Furthermore, C#minor is classically structured in functions, statements, and expressions. Expressions have no side effects; they include reading temporary variables (which do not reside in memory), taking the address of a non-temporary variable, constants, arithmetic operations, and dereferencing addresses. The arithmetic, logical, comparison, and conversion operators are roughly those of C, but without overloading: for example, distinct operators are provided for integer multiplication and floating-point multiplication. Likewise, there are no implicit casts: all conversions between numerical types are explicit. Statements offer both structured control and `goto` with labels. C loops as well as `break` and `continue` statements are encoded as infinite loops with a multi-level `exit` n that jumps to the end of the $(n+1)$-th enclosing `block`.

3. Verifying Constant-Time Security

Our static analyzer operates over C#minor programs. In this paper, we use a simpler While toy-language for clarity. It is defined in the first part of this section. Then, we detail our model for constant-time leakage, and explain the tainting semantics we have defined to track data dependencies in programs. Last, we explain the main algorithm of our static analyzer, and detail its proof of correctness.

Expressions: $e ::= n \mid a \mid x \mid e_1 \oplus e_2 \quad \oplus \in \{+, -, \times, /, =, <, >\}$

Statements: $p ::= \texttt{skip} \mid *e_1 \leftarrow e_2 \mid x \leftarrow e \mid x \leftarrow *e \mid p_1 ; p_2$
$\mid \texttt{if } e \texttt{ then } p_1 \texttt{ else } p_2 \mid \texttt{while } e \texttt{ do } p$

Figure 2. Syntax of While programs

3.1. The While Language

Our While language is classically structured in statements and expressions, as shown in Figure 2. Expressions include integer constants, array identifiers, variable identifiers, arithmetic operations and tests. Statements include skip statements, stores $*x \leftarrow y$, loads $x \leftarrow *y$, assignments $x \leftarrow y$, sequences, if and while statements.

Our While language is peculiar as it supports arrays in order to model memory aliasing. We will mainly use a for array identifiers and x for variable identifiers. As an example, the program $x \leftarrow a{+}2; y \leftarrow *(x{+}3)$ first starts by assigning the value $a + 2$ to variable x and then loads the value at offset 5 of the array a into the variable y. In this example, $x - 2$ is an alias of a.

The semantics of While is defined in Figure 3 using a small-step style for statements and a big-step style for expressions, supporting the reasoning on non-terminating programs. Contrary to the C language, the semantics is deterministic (and so is the semantics of C#minor).

A location, usually named l, is a pair of an array identifier and an offset represented by a positive integer. A value v can either be a location or an integer. An environment σ is a pair $(\sigma_{\mathbb{X}}, \sigma_{\mathbb{A}})$ composed of a partial map from variables in set $\mathbb{X}$ of variable identifiers to values and a partial map from memory locations $\mathbb{A} \times \mathbb{N}$ to values where $\mathbb{A}$ is a set of array identifiers and values $\mathbb{V}$ are either locations or integers. We will write $\sigma(x)$ to mean $\sigma_{\mathbb{X}}(x)$ and $\sigma(l)$ for $\sigma_{\mathbb{A}}(l)$.

Given an environment σ, an expression e evaluates to a value v (written $\langle \sigma, e \rangle \rightarrow v$). A constant is interpreted as an integer. An array identifier a evaluates to its location, it is equivalent to writing $a + 0$. To evaluate a variable, its value is looked up in the environment σ and more precisely in its $\sigma_{\mathbb{X}}$ component. Finally, to evaluate $e_1 \oplus e_2$, it is simply needed to evaluate e_1 and e_2 separately and combine the resulting values by interpreting $\oplus$ into its corresponding operator $\boxplus$, where $\boxplus$ is the usual semantics of the operator $\oplus \in \{+, -, \times, /, =, <, >\}$. For example, $e_1 = e_2$ returns 0 if the test is false, and 1 otherwise.

The execution of a statement s results in an updated state with a new environment σ' and a new statement to execute s', written $\langle \sigma, s \rangle \rightarrow \langle \sigma', s' \rangle$. We write $\sigma(e)$ to denote the value of expression e in state σ (i.e., $\langle \sigma, e \rangle \rightarrow \sigma(e)$) and we use $\sigma[l \mapsto v]$ to denote the environment that behaves the same as σ except that it returns value v for location l. We consider all arrays to be of finite size and initially declared, similarly to global variables in C. Thus, $\sigma(x, n)$ and $\sigma[(a, n) \mapsto v]$ may fail either because a is not a valid array name or because it is an out-of-bound access. $\sigma(l) = v$ means that l is a valid location for σ, whereas $\sigma(l) = \bot$ means the opposite. Similarly, $\sigma[l \mapsto v] = \sigma'$ indicates the success of the update. We assume a memory model similar to C's except that variables have no addresses but behave more like registers.

To execute a store $*e_1 \leftarrow e_2$, it is first needed for e_1 to evaluate into a location l and e_2 to evaluate into a value v; the environment is then updated so that location l maps to v. Similarly, to execute a load $x \leftarrow *e$, the expression e must first evaluate into a location l. It is then needed to retrieve its corresponding value v in the environment and update the environment so that x maps to v. To execute the assignment $x \leftarrow e$, it is only needed to evaluate e and update the environment so that x maps to the resulting value. To execute a sequence $p_1; p_2$, either p_1 is a skip and p_2 is the only statement left to execute, or we first need

$$l \in \mathbb{L} = \mathbb{A} \times \mathbb{N} \quad v \in \mathbb{V} = \mathbb{L} + \mathbb{Z}$$

$$\sigma = (\sigma_{\mathbb{X}}, \sigma_{\mathbb{A}}) \in \mathbb{M} = (\mathbb{X} \to \mathbb{V} \cup \{\bot\}) \times (\mathbb{L} \to \mathbb{V} \cup \{\bot\})$$

$$\frac{}{\langle \sigma, n \rangle \to n}$$

$$\frac{}{\langle \sigma, a \rangle \to (a, 0)}$$

$$\frac{\sigma(x) = v}{\langle \sigma, x \rangle \to v}$$

$$\frac{\langle \sigma, e_1 \rangle \to v_1 \qquad \langle \sigma, e_2 \rangle \to v_2}{\langle \sigma, e_1 \oplus e_2 \rangle \to v_1 \boxplus v_2}$$

$$\text{store}\ \frac{\langle \sigma, e_1 \rangle \to l \qquad \langle \sigma, e_2 \rangle \to v \qquad \sigma[l \mapsto v] = \sigma'}{\langle \sigma, *e_1 \leftarrow e_2 \rangle \to \langle \sigma', \texttt{skip} \rangle}$$

$$\text{load}\ \frac{\langle \sigma, e \rangle \to l \qquad \sigma(l) = v \qquad \sigma[x \mapsto v] = \sigma'}{\langle \sigma, x \leftarrow *e \rangle \to \langle \sigma', \texttt{skip} \rangle}$$

$$\text{assign}\ \frac{\langle \sigma, e \rangle \to v \qquad \sigma[x \mapsto v] = \sigma'}{\langle \sigma, x \leftarrow e \rangle \to \langle \sigma', \texttt{skip} \rangle}$$

$$\text{skipseq}\ \frac{}{\langle \sigma, \texttt{skip}\,; p \rangle \to \langle \sigma, p \rangle}$$

$$\text{seq}\ \frac{\langle \sigma, p_1 \rangle \to \langle \sigma', p_1' \rangle}{\langle \sigma, p_1\,; p_2 \rangle \to \langle \sigma', p_1'\,; p_2 \rangle}$$

$$\text{iftrue}\ \frac{\langle \sigma, e \rangle \to \texttt{true}}{\langle \sigma, \texttt{if}\ e\ \texttt{then}\ p_1\ \texttt{else}\ p_2 \rangle \to \langle \sigma, p_1 \rangle}$$

$$\text{iffalse}\ \frac{\langle \sigma, e \rangle \to \texttt{false}}{\langle \sigma, \texttt{if}\ e\ \texttt{then}\ p_1\ \texttt{else}\ p_2 \rangle \to \langle \sigma, p_2 \rangle}$$

$$\text{whiletrue}\ \frac{\langle \sigma, e \rangle \to \texttt{true}}{\langle \sigma, \texttt{while}\ e\ \texttt{do}\ p \rangle \to \langle \sigma, p\,; \texttt{while}\ e\ \texttt{do}\ p \rangle}$$

$$\text{whilefalse}\ \frac{\langle \sigma, e \rangle \to \texttt{false}}{\langle \sigma, \texttt{while}\ e\ \texttt{do}\ p \rangle \to \langle \sigma, \texttt{skip} \rangle}$$

Figure 3. Semantics of While programs

to execute p_1, resulting in a new state $\langle \sigma', p_1' \rangle$. Then, $p_1'; p_2$ is left to execute in the new environment σ'. Classically, in order to execute a conditional branching `if e then p1 else p2`, it is needed to evaluate e and execute accordingly the appropriate branch. Similarly, a loop `while e do p` stops if e evaluates to false and continues otherwise.

As a remark, the evaluation of an expression can only be stuck in two ways, either because it is trying to retrieve the value of an undefined variable (i.e., $\sigma(x)$ fails when x is not defined in σ), or because $v_1 \boxplus v_2$ is not defined (e.g., because of a division by 0). Finally, the execution of statements can only be stuck when the semantic rule evaluates an expression and gets stuck, or the corresponding result has the

wrong value type, or the result is a non-valid location. For instance, a branching statement cannot branch on a location value, or $\sigma(l)$ fails because it is an out-of-bound access or there is no associated value yet in the environment.

The reflexive transitive closure of this small-step semantics represents the execution of a program. When the program terminates (resp. diverges, e.g. when an infinite loop is executed), it is a finite (resp. infinite) execution of steps. The execution of a program is *safe* iff either the program terminates (i.e., its final semantic state is $\langle \sigma, \texttt{skip} \rangle$, meaning that there is no more statement to execute) or the program diverges. The execution of a program is *stuck* on $\langle \sigma, s \rangle$ when s differs from `skip` and no semantic rule can be applied. A program is *safe* when all of its executions are safe. We write $(\langle \sigma_i, p_i \rangle)_i$ for the execution $\langle \sigma_0, p_0 \rangle \to \langle \sigma_1, p_1 \rangle \to \ldots$ of program p_0 with initial environment σ_0.

3.2. Constant-Time Security

In our model, we assume that branching statements and memory accesses may leak information through their execution. We use a similar definition of constant-time security to the one given in [20]. We define a *leakage model* $\mathscr{L}$ as a map from semantic states $\langle \sigma, p \rangle$ to sequences of observations $\mathscr{L}(\langle \sigma, p \rangle)$ with ε being the empty observation. Two executions are said to be *indistinguishable* when their observations are the same:

$$\mathscr{L}(\langle \sigma_0, p_0 \rangle) \cdot \mathscr{L}(\langle \sigma_1, p_1 \rangle) \cdot \ldots = \mathscr{L}(\langle \sigma'_0, p'_0 \rangle) \cdot \mathscr{L}(\langle \sigma'_1, p'_1 \rangle) \cdot \ldots.$$

Definition 1 (Constant-time leakage model)**.** *Our leakage model is such that the following equalities hold.*

(1) $\mathscr{L}(\langle \sigma, \texttt{if e then } p_1 \texttt{ else } p_2 \rangle) = \sigma(e)$
(2) $\mathscr{L}(\langle \sigma, \texttt{while e do } p \rangle) = \sigma(e)$
(3) $\mathscr{L}(\langle \sigma, *e_1 \leftarrow e_2 \rangle) = \sigma(e_1)$
(4) $\mathscr{L}(\langle \sigma, x \leftarrow *e \rangle) = \sigma(e)$
(5) $\mathscr{L}(\langle \sigma, p_1; p_2 \rangle) = \mathscr{L}(\langle \sigma, p_1 \rangle)$
(6) $\mathscr{L}(\langle \sigma, p \rangle) = \varepsilon$ *otherwise*

The first and second lines mean that the value of branching conditions is considered as leaked. The third and fourth lines mean that the address of a store and load access is also considered as leaked. The fifth line explains that a sequence leaks exactly what is leaked by the first part of the sequence; this is due to the semantics of sequence which depends on the execution of the first statement. Finally, the other statements produce a silent observation. Given this leakage model, the following lemma follows.

Lemma 1 (Same control-flow)**.** *If* $(\langle \sigma_1, p_1 \rangle) \to (\langle \sigma_2, p_2 \rangle)$ *and* $(\langle \sigma'_1, p'_1 \rangle) \to (\langle \sigma'_2, p'_2 \rangle)$ *such that* $p_1 = p'_1$ *and* $\mathscr{L}(\langle \sigma_1, p_1 \rangle) = \mathscr{L}(\langle \sigma'_1, p'_1 \rangle)$ *then* $p_2 = p'_2$*.*

Proof. By induction on $(\langle \sigma_1, p_1 \rangle) \to (\langle \sigma_2, p_2 \rangle)$:

- In the assign, store and load cases, $p_2 = p'_2 = \texttt{skip}$.
- In the skipseq case, there exists p such that $p_1 = p'_1 = \texttt{skip; } p$ and thus $p_2 = p'_2 = p$.
- In the seq case, there exists q_1, q'_1, q''_1 and q_2 such that $p_1 = p'_1 = q_1; q_2$ and $(\langle \sigma_1, q_1 \rangle) \to (\langle \sigma_2, q'_1 \rangle)$ and $(\langle \sigma'_1, q_1 \rangle) \to (\langle \sigma'_2, q''_1 \rangle)$. In order to use the induction hypothesis to prove $q'_1 = q''_1$, we first need to prove that $\mathscr{L}(\langle \sigma_1, q_1 \rangle) = \mathscr{L}(\langle \sigma'_1, q_1 \rangle)$. This is true by definition since $\mathscr{L}(\langle \sigma_1, p_1 \rangle) =$

$\mathscr{L}(\langle\sigma_1, q_1; q_2\rangle) = \mathscr{L}(\langle\sigma_1, q_1\rangle)$ and also $\mathscr{L}(\langle\sigma'_1, p'_1\rangle) = \mathscr{L}(\langle\sigma'_1, q_1; q_2\rangle) = \mathscr{L}(\langle\sigma'_1, q_1\rangle)$ and $\mathscr{L}(\langle\sigma_1, p_1\rangle) = \mathscr{L}(\langle\sigma'_1, p'_1\rangle)$. Thus, since $p_2 = q'_1; q_2$ and $p'_2 = q''_1; q_2$, we have finally $p_2 = p'_2$.

- In the iftrue, iffalse, whiletrue and whilefalse cases, we simply use $\mathscr{L}(\langle\sigma_1, p_1\rangle) = \mathscr{L}(\langle\sigma'_1, p'_1\rangle)$ to justify that the same branch is taken.

□

Finally, the following theorem follows by induction.

Theorem 1. *Two indistinguishable executions of a program necessarily have the same control flow.*

Proof. Suppose we have two indistinguishable executions $(\langle\sigma_i, p_i\rangle)_i$ and $(\langle\sigma'_i, p'_i\rangle)_i$ such that $p_0 = p'_0$. We prove by induction on i that for all i, $p_i = p'_i$.

- It's true by hypothesis for $i = 0$.
- Suppose that $p_i = p'_i$.
  - If the execution is stuck for $\langle\sigma_i, p_i\rangle$, then necessarily it is because p_i tries to write or read an invalid location (i.e. the value is not a location but a constant or it is an out-of-bound location) or it tries to branch on a non-boolean value. However, by definition of indistinguishability and the leakage model, these values must be the same in both executions, thus the execution is also stuck for $\langle\sigma'_i, p'_i\rangle$.
  - Symmetrically, if the execution is stuck for $\langle\sigma'_i, p'_i\rangle$, it is also stuck for $\langle\sigma_i, p_i\rangle$.
  - If $\langle\sigma_i, p_i\rangle \to \langle\sigma_{i+1}, p_{i+1}\rangle$, then there exists σ'_{i+1}, p'_{i+1} such that $\langle\sigma'_i, p'_i\rangle \to \langle\sigma'_{i+1}, p'_{i+1}\rangle$ or both executions would have been stuck. By using the previous lemma, we prove that $p_{i+1} = p'_{i+1}$.

Both executions have thus the same control flow. □

Given a program, we assume that the attacker has access to the values of some of its inputs, which we call the *public* input array variables, and does not have access of the other ones, which we call the *secret* input array variables. Given a set X of array names, and two environments σ and σ', we say that σ and σ' are X-equivalent if σ and σ' both share the same public input values. Two executions $(\langle\sigma_i, p_i\rangle)_i$ and $(\langle\sigma'_i, p'_i\rangle)_i$ are initially X-equivalent if σ_0 and σ'_0 are X-equivalent.

Definition 2 (Constant-time security)**.** *A program p is constant time if for any set X_i of public input array variables, all of its initially X_i-equivalent executions are indistinguishable.*

This definition means that a constant-time program is such that, any pair of its executions that only differ on its secrets must leak the exact same information. This also gives a definition of constant-time security for infinite execution.

3.3. Reducing Security to Safety

In order to prove that a program satisfy constant-time security as defined in Definition 2, we reduce the problem to checking whether the program is safe in a different semantics. The issue is thus twofold, we first need to prove that safety in this instrumented semantics implies constant-time security in the standard semantics and second, we need to design an analyzer for this second semantics. This can also be obtained

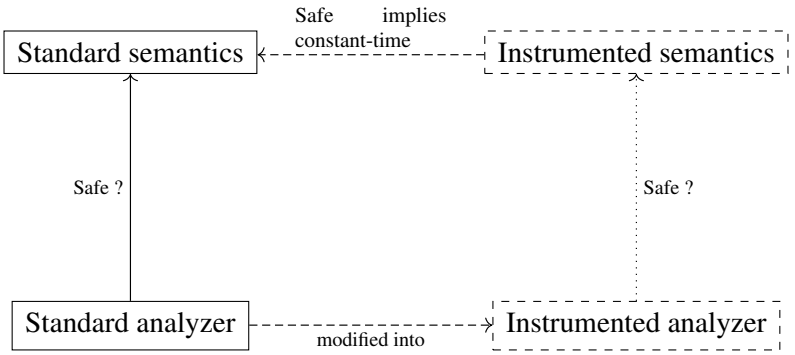

Figure 4. Methodology

by modifying an analyzer for the standard semantics as illustrated by Figure 4. Plain lines indicate what we assume to already have, while dashed lines indicates what needs to be designed or proved.

We introduce an intermediate tainting semantics for While programs in Figure 5, and use the $\rightsquigarrow$ symbol to distinguish its executions from those of the original semantics. The tainting semantics is an instrumentation of the While semantics that tracks dependencies related to secret values. In the tainted semantics, a program gets stuck if branchings or memory accesses depend on secrets. We introduce taints, either $\mathcal{H}$ (High) or $\mathcal{L}$ (Low) to respectively represent secret and public values and a union operator on taints defined as follows: $\mathcal{L} \sqcup \mathcal{L} = \mathcal{L}$ and for all t, $\mathcal{H} \sqcup t = t \sqcup \mathcal{H} = \mathcal{H}$. It is used to compute the taint of a binary expression. In the instrumented semantics, we take into account taints in semantic values: the semantic state σ becomes a tainted state $\overline{\sigma}$, where locations are now mapped to pairs made of a value and a taint.

Let us note that for a dereferencing expression $*e$ to have a value, the taint associated to e must be $\mathcal{L}$. Indeed, we forbid memory read accesses that might leak secret values. This concerns dereferencing expressions (loads) and assignment statements. Similarly, test conditions in branching statements must also have a $\mathcal{L}$ taint.

The instrumented semantics strictly forbids more behaviors than the regular semantics (defined in Figure 3) as shown by the following lemma.

Lemma 2. *Any execution $(\langle\overline{\sigma_i}, p_i\rangle)_i$ of program p_0 in the tainting semantics implies that $(\langle\sigma_i, p_i\rangle)_i$ is an execution of p_0 in the regular semantics where for all i, $\sigma_i = \mathcal{E} \circ \overline{\sigma_i}$ and $\mathcal{E}(a,b) = a$ for all pairs (a,b) is an erasure function.*

Proof. For all $\overline{\sigma}$, $\overline{\sigma}'$, p, p' such that $\langle\overline{\sigma}, p\rangle \to \langle\overline{\sigma}', p'\rangle$, we can easily prove by immediate induction that $\langle\sigma, p\rangle \to \langle\sigma', p'\rangle$ where $\sigma = \mathcal{E} \circ \overline{\sigma}$ and $\sigma' = \mathcal{E} \circ \overline{\sigma}'$.

Finally, by induction on the execution and using this lemma, the theorem is easily proven. □

However, the converse is not necessarily true. For instance, suppose that variable x contains a secret value. Then, $*(a+x) \leftarrow 2$ is not safe in the instrumented semantics because $a+x$ has taint $\mathcal{H}$, while it is safe in the regular semantics provided that $a+x$ corresponds to a valid location.

An immediate consequence of the lemma is that the instrumented semantics preserves the regular behavior of programs, as stated by the following theorem.

$$t \in \mathbb{T} = \{\mathcal{L}, \mathcal{H}\}$$

$$\overline{\mathbb{V}} = \mathbb{V} \times \mathbb{T}$$

$$\overline{\sigma} = (\overline{\sigma_{\mathbb{X}}}, \overline{\sigma_{\mathbb{A}}}) \in \overline{\mathbb{M}} = (\mathbb{X} \to \overline{\mathbb{V}} \cup \{\bot\}) \times (\mathbb{L} \to \overline{\mathbb{V}} \cup \{\bot\})$$

$$\frac{}{\langle \overline{\sigma}, n \rangle \rightsquigarrow (n, \mathcal{L})}$$

$$\frac{}{\langle \overline{\sigma}, a \rangle \rightsquigarrow ((a, 0), \mathcal{L})}$$

$$\frac{\overline{\sigma}(x) = (v, t)}{\langle \overline{\sigma}, x \rangle \rightsquigarrow (v, t)}$$

$$\frac{\langle \overline{\sigma}, e_1 \rangle \rightsquigarrow (v_1, t_1) \qquad \langle \overline{\sigma}, e_2 \rangle \rightsquigarrow (v_2, t_2)}{\langle \overline{\sigma}, e_1 \oplus e_2 \rangle \rightsquigarrow (v_1 \boxplus v_2, t_1 \sqcup t_2)}$$

$$\frac{\langle \overline{\sigma}, e_1 \rangle \rightsquigarrow (l, \mathcal{L}) \qquad \langle \overline{\sigma}, e_2 \rangle \rightsquigarrow (v, t) \qquad \overline{\sigma}[l \mapsto (v, t)] = \sigma'}{\langle \overline{\sigma}, *e_1 \leftarrow e_2 \rangle \rightsquigarrow \langle \overline{\sigma}', \texttt{skip} \rangle}$$

$$\frac{\langle \overline{\sigma}, e \rangle \rightsquigarrow (l, \mathcal{L}) \qquad \overline{\sigma}(l) \rightsquigarrow (v, t) \qquad \overline{\sigma}[x \mapsto (v, t)] = \sigma'}{\langle \overline{\sigma}, x \leftarrow *e \rangle \rightsquigarrow \langle \overline{\sigma}', \texttt{skip} \rangle}$$

$$\frac{\langle \overline{\sigma}, e \rangle \to (v, t) \qquad \overline{\sigma}[x \mapsto (v, t)] = \overline{\sigma}'}{\langle \overline{\sigma}, x \leftarrow e \rangle \to \langle \overline{\sigma}', \texttt{skip} \rangle}$$

$$\frac{}{\langle \overline{\sigma}, \texttt{skip}\,; p \rangle \rightsquigarrow \langle \overline{\sigma}, p \rangle}$$

$$\frac{\langle \overline{\sigma}, p_1 \rangle \rightsquigarrow \langle \overline{\sigma}', p_1' \rangle}{\langle \overline{\sigma}, p_1 ; p_2 \rangle \rightsquigarrow \langle \overline{\sigma}', p_1' ; p_2 \rangle}$$

$$\frac{\langle \overline{\sigma}, e \rangle \rightsquigarrow (\texttt{true}, \mathcal{L})}{\langle \overline{\sigma}, \texttt{if } e \texttt{ then } p_1 \texttt{ else } p_2 \rangle \rightsquigarrow \langle \overline{\sigma}, p_1 \rangle}$$

$$\frac{\langle \overline{\sigma}, e \rangle \rightsquigarrow (\texttt{false}, \mathcal{L})}{\langle \overline{\sigma}, \texttt{if } e \texttt{ then } p_1 \texttt{ else } p_2 \rangle \rightsquigarrow \langle \overline{\sigma}, p_2 \rangle}$$

$$\frac{\langle \overline{\sigma}, e \rangle \rightsquigarrow (\texttt{true}, \mathcal{L})}{\langle \overline{\sigma}, \texttt{while } e \texttt{ do } p \rangle \rightsquigarrow \langle \overline{\sigma}, p ; \texttt{while } e \texttt{ do } p \rangle}$$

$$\frac{\langle \overline{\sigma}, e \rangle \rightsquigarrow (\texttt{false}, \mathcal{L})}{\langle \overline{\sigma}, \texttt{while } e \texttt{ do } p \rangle \rightsquigarrow \langle \overline{\sigma}, \texttt{skip} \rangle}$$

Figure 5. Tainting semantics for While programs

Theorem 2. *Any safe execution $(\langle \overline{\sigma_i}, p_i \rangle)_i$ of program p_0 in the tainting semantics implies that the execution $(\langle \sigma_i, p_i \rangle)_i$ is also safe in the regular semantics.*

As an immediate corollary, any safe program according to the tainting semantics is also safe according to the regular semantics.

Proof. Let $(\langle \overline{\sigma_i}, p_i \rangle)_i$ be a safe execution of p_0 in the tainting semantics. As it is a safe execution, it either

diverges or terminates.

- If $(\langle\overline{\sigma_i}, p_i\rangle)_i$ is diverging (i.e. infinite), then so is $(\langle\sigma_i, p_i\rangle)_i$ thanks to the previous lemma.
- If $(\langle\overline{\sigma_i}, p_i\rangle)_i$ is terminating, then there exists some n such that $p_n = \texttt{skip}$, therefore $(\langle\sigma_i, p_i\rangle)_{i\leqslant n}$ is also terminating.

$(\langle\sigma_i, p_i\rangle)_i$ is a safe execution in the regular semantics. □

Theorem 2 is useful to prove our main theorem relating our instrumented semantics and the constant-time property we want to verify on programs.

Theorem 3. *Any safe program w.r.t. the tainting semantics is constant time.*

Proof. Let p_0 be a safe program with respect to the tainting semantics. Let X_i be a set of public variables and let $(\langle\sigma_i, p_i\rangle)_i$ and $(\langle\sigma'_i, p'_i\rangle)_i$ be two safe executions of p_0 that are initially X_i-equivalent.

We now need to prove that both executions are indistinguishable. Let $\overline{\sigma_0}$ be such that for all $x \in X_i,\ n \in \mathbb{N},\ \overline{\sigma_0}(x,n) = (\sigma_0(x,n), \mathcal{L})$ and also for all $x \notin X_i,\ n \in \mathbb{N},\ \overline{\sigma_0}(x,n) = (\sigma_0(x,n), \mathcal{H})$.

By safety of program p_0 according to the tainting semantics, there exists some states $\overline{\sigma_1}, \overline{\sigma_2}, \ldots$ such that $\langle\overline{\sigma_0}, p_0\rangle \rightsquigarrow \langle\overline{\sigma_1}, p_1\rangle \rightsquigarrow \ldots$ is a safe execution. Let $\sigma_{n'} = \mathcal{E} \circ \overline{\sigma_n}$, we prove by strong induction on n that $\sigma_{n'} = \sigma_n$.

- It is clearly true for $n = 0$ by definition of $\overline{\sigma_0}$.
- Suppose it is true for all $k < n$ and let us prove it for n. By using theorem 2, we know that there exists a safe execution $\langle\sigma_0, p_0\rangle \rightarrow \langle\sigma_{1'}, p_{1'}\rangle \rightarrow \ldots \rightarrow \langle\sigma_{n'}, p_{n'}\rangle \rightarrow \ldots$. Furthermore, the semantics is deterministic and we know that $\langle\sigma_0, p_0\rangle \rightarrow \langle\sigma_1, p_1\rangle \rightarrow \ldots$. Therefore, we have the following series of equalities: $\sigma_{1'} = \sigma_1, p_{1'} = p_1, \ldots \sigma_{n'} = \sigma_n, p_{n'} = p_n$.

Thus, for all $k \in \mathbb{N}$, the state $\overline{\sigma_k}$ verifies $\sigma_k = \mathcal{E} \circ \overline{\sigma_k}$. Similarly, we define $\overline{\sigma_0}', \overline{\sigma_1}', \ldots$ for the second execution which also satisfies the same property by construction.

Finally, we need to prove that for all $n \in \mathbb{N}$, $L(\langle\sigma_n, p_n\rangle) = L(\langle\sigma'_n, p'_n\rangle)$.

First, we informally define the notation $\sigma_n =_{\mathcal{L}} \sigma'_n$ for all $n \in \mathbb{N}$ as $\overline{\sigma}_n$ and $\overline{\sigma}'_n$, as previously defined, agree on the taints of both variables and locations, and if the taint is $\mathcal{L}$, then they also agree on the value. Formally, this means that for all r where r is either a location l or a variable x, either $\overline{\sigma}_n(r)$ and $\overline{\sigma}'_n(r)$ are undefined, or there exists a taint t such that $\overline{\sigma}_n(r) = (\sigma_n(r), t)$ and $\overline{\sigma}'_n(r) = (\sigma'_n(r), t)$ and if $t = \mathcal{L}$, then $\sigma_n(r) = \sigma'_n(r)$.

Second, we prove by induction on e that for all n and e that if $\sigma_n =_{\mathcal{L}} \sigma'_n$, $\langle\overline{\sigma_n}, e\rangle \rightsquigarrow (v,t)$ and $\langle\overline{\sigma'_n}, e\rangle \rightsquigarrow (v',t')$, then $t = t'$ and if $t = t' = \mathcal{L}$, then $v = v'$.

- This is trivially true if $e = n$ or $e = a$.
- If $e = x$, then it is true by definition of $\sigma_n =_{\mathcal{L}} \sigma'_n$
- If $e = e_1 \oplus e_2$, then we apply the induction hypotheses on $\langle\overline{\sigma_n}, e_1\rangle \rightsquigarrow (v_1, t_1)$ and $\langle\overline{\sigma_n}', e_1\rangle \rightsquigarrow (v'_1, t'_1)$ and on $\langle\overline{\sigma_n}, e_2\rangle \rightsquigarrow (v_2, t_2)$ and $\langle\overline{\sigma_n}', e_2\rangle \rightsquigarrow (v'_2, t'_2)$. Since $t = t_1 \sqcup t_2$ and $t' = t'_1 \sqcup t'_2$ and $t_1 = t'_1$ and $t_2 = t'_2$, we have that $t = t'$. If $t = t' = \mathcal{L}$, then $t_1 = t'_1 = \mathcal{L}$ and $t_2 = t'_2 = \mathcal{L}$, thus $v = v_1 \boxplus v_2 = v'_1 \boxplus v'_2 = v'$.

This lemma is thus proven.

Finally, for all $n \in \mathbb{N}$, let us prove by induction on p_n that if $p_n = p'_n$ and $\sigma_n =_{\mathcal{L}} \sigma'_n$, then $p_{n+1} = p'_{n+1}$ and $\sigma_{n+1} =_{\mathcal{L}} \sigma'_{n+1}$.

- If $p_n = \texttt{skip; } p'$, it is true because $p_{n+1} = p'_{n+1} = p'$, $\sigma_{n+1} = \sigma_n$ and also $\sigma'_{n+1} = \sigma'_n$.
- If $p_n = p\texttt{; } p'$, it is true by induction hypothesis.
- If $p_n = \texttt{if } e \ldots$ or $p_n = \texttt{while } e \ldots$, we have $\sigma_{n+1} = \sigma_n$ and $\sigma'_{n+1} = \sigma'_n$. Furthermore, we know that there exists some v such that $\langle \overline{\sigma_n}, e\rangle \rightsquigarrow (v, \mathcal{L})$ and similarly, there exists v' such that $\langle \overline{\sigma_n}', e\rangle \rightsquigarrow (v', \mathcal{L})$ because of the safety in the tainting semantics. Since $\sigma_n(e) = v$, $\sigma'_n(e) = v'$ and $\sigma_n =_{\mathcal{L}} \sigma'_n$, we have $v = v'$ by using the previous lemma and thus $p_{n+1} = p'_{n+1}$.
- If $p_n = x \leftarrow *e$, we can prove as previously that $\sigma_n(e_1) = \sigma'_n(e_1) = l$. Furthermore, we have $p_{n+1} = p'_{n+1} = \texttt{skip}$. It is left to prove that $\sigma_{n+1} =_{\mathcal{L}} \sigma'_{n+1}$. If $\overline{\sigma_n}(l) = (v, t)$ and $\overline{\sigma_n}'(l) = (v', t')$, then $t = t'$ since $\sigma_n =_{\mathcal{L}} \sigma'_n$. If $t = t' = \mathcal{L}$, then $v = v'$ and $\sigma_{n+1} = \sigma_n[x \mapsto v]$ and $\sigma'_{n+1} = \sigma'_n[x \mapsto v']$, thus $\sigma_{n+1} =_{\mathcal{L}} \sigma'_{n+1}$. Similarly, if $t = t' = \mathcal{H}$, then $\sigma_{n+1} =_{\mathcal{L}} \sigma'_{n+1}$.
- If $p_n = x \leftarrow e$, we know that $p_{n+1} = p'_{n+1} = \texttt{skip}$. Furthermore, there exists v, v', t, t' such that $\langle \overline{\sigma_n}, e\rangle \rightsquigarrow (v, t)$ and $\langle \overline{\sigma_n}', e\rangle \rightsquigarrow (v', t')$. By using the previous lemma, we know that $t = t'$, and if $t = t' = \mathcal{L}$, then $v = v'$. Thus $\sigma_{n+1} = \sigma_n[x \mapsto v] =_{\mathcal{L}} \sigma'_n[x \mapsto v'] = \sigma'_{n+1}$.
- If $p_n = *e_1 \leftarrow e_2$, we have $p_{n+1} = p'_{n+1} = \texttt{skip}$. By using the same reasoning as previously, we can prove that $\sigma_n(e_1) = \sigma'_n(e_1) = l$. There exists v, v', t, t' such that $\langle \overline{\sigma_n}, e_2\rangle \rightsquigarrow (v, t)$ and $\langle \overline{\sigma_n}', e_2\rangle \rightsquigarrow (v', t')$ and thus $\sigma_{n+1} = \sigma_n[l \mapsto v]$ and $\sigma'_{n+1}[l \mapsto v']$. By using the previous lemma, we know that $t = t'$ and if $t = t' = \mathcal{L}$, then $v = v'$ and $\sigma_{n+1} =_{\mathcal{L}} \sigma'_{n+1}$. If $t = t' = \mathcal{H}$, then $\sigma_{n+1} =_{\mathcal{L}} \sigma'_{n+1}$ by definition.

Finally, by exploiting this lemma, an induction proves that for all $n \in \mathbb{N}$, $p_n = p'_n$ and $\sigma_n =_{\mathcal{L}} \sigma'_n$. Furthermore, a direct consequence is that for all $n \in \mathbb{N}$, $L(\langle \sigma_n, p_n\rangle) = L(\langle \sigma'_n, p'_n\rangle)$ and thus both executions are indistinguishable: the program is constant time. □

3.4. Abstract Interpreter

To prove that a program is safe according to the tainting semantics, we design a static analyzer based on abstract interpretation. It computes a correct approximation of the execution of the analyzed program, thus if the approximative execution is safe, then the actual execution must necessarily be safe.

Similarly to how we built a tainting semantics from a regular semantics, we explain how to modify an abstract interpreter for the regular semantics into an abstract interpreter for the tainting semantics. First, we suppose that the regular abstract interpreter provides a domain of abstract values $\mathbb{V}^\sharp$ that supports an operator $\mathsf{concretize}^\sharp : \mathbb{V}^\sharp \to \mathcal{P}(\mathbb{V})$ which takes an abstract value and returns the concrete values represented by the abstract value. We also suppose that the abstract interpreter provides $\mathbb{M}^\sharp$, an abstraction of concrete environments that maps locations and variables to values. We do not need nor want to know exactly how $\mathbb{M}^\sharp$ is defined, as it might use relational definitions which are quite complex. We only need to use $\mathbb{M}^\sharp$ to modify the abstract analyzer.

Finally, we suppose that the abstract analyzer provides the following abstract operators:

- $\mathsf{eval}^\sharp : \mathbb{M}^\sharp \to \mathsf{expr} \to \mathbb{V}^\sharp$ takes an abstract environment, an expression and evaluates it in the abstract environment and returns the corresponding abstract value;
- $\mathsf{assign}^\sharp : \mathbb{M}^\sharp \to \mathbb{X} \to \mathsf{expr} \to \mathbb{M}^\sharp$ takes an abstract environment, a variable identifier, an expression and models an assignment to a variable;
- $\mathsf{store}^\sharp : \mathbb{M}^\sharp \to \mathsf{expr} \to \mathsf{expr} \to \mathbb{M}^\sharp$ takes an abstract environment and two expression e_1 and e_2 and models $*e_1 \leftarrow e_2$;
- $\mathsf{load}^\sharp : \mathbb{M}^\sharp \to \mathbb{X} \to \mathsf{expr} \to \mathbb{M}^\sharp$ takes an abstract environment, a variable identifier, an expression and models a load $x \leftarrow *e$;

Figure 6. Abstract taint lattice

- $\mathsf{assert}^\sharp : \mathbb{M}^\sharp \to \mathsf{expr} \to \mathbb{M}^\sharp$ takes an abstract environment, an expression and returns an abstract environment where the expression is true. This is useful when analyzing a branching condition such as $x < 5$, if we know beforehand that $x \in [0, 42]$, we can restrict x to $[0, 4]$ in the "then" branch, and restrict it to $[5, 42]$ in the "else" branch.

The abstract operators form an interface that is parameterized by $\mathbb{V}^\sharp$ and $\mathbb{M}^\sharp$ that we will name $\mathsf{AbMem}(\mathbb{V}^\sharp, \mathbb{M}^\sharp)$.

Now, in order for the analyzer to handle the tainting semantics, we need to introduce an abstraction of taints $\mathbb{T}^\sharp = \{\mathcal{L}^\sharp, \mathcal{H}^\sharp\}$ which forms a lattice represented in Figure 6. We will use $\mathcal{L}^\sharp$ to indicate a value that has exactly taint $\mathcal{L}$ while $\mathcal{H}^\sharp$ indicates that a value may have taint $\mathcal{L}$ or $\mathcal{H}$. In order to analyze the following snippet, it is necessary to correctly approximate the taint of the value that will be assigned to variable x after execution.

```
if /* low expr */
  x ← /* high expr */
else
  x ← /* low expr */
```

As it can either be $\mathcal{L}$ or $\mathcal{H}$, we use the approximation $\mathcal{H}^\sharp$. We could have used $\mathcal{H}^\sharp$ to indicate that a variable or location can only have a $\mathcal{H}$ value, however constant-time security is not interested in knowing that value has exactly $\mathcal{H}$ taint, but only in knowing that it *may* have a $\mathcal{H}$ taint.

Now, we explain how to modify the analyzer so that it can track abstract taints, this effectively forms a functor from the previous interface $\mathsf{AbMem}(\mathbb{V}^\sharp, \mathbb{M}^\sharp)$ to a new interface $\mathsf{AbMem}(\overline{\mathbb{V}}^\sharp, \overline{\mathbb{M}}^\sharp)$ that can track abstract taints where $\overline{\mathbb{V}}^\sharp = \mathbb{V}^\sharp \times \mathbb{T}^\sharp$ and $\overline{\mathbb{M}}^\sharp = \mathbb{M}^\sharp \times ((\mathbb{X} + \mathbb{L}) \to \mathbb{T}^\sharp)$.

We first start by defining $\overline{\mathsf{taint}}^\sharp : \overline{\mathbb{M}}^\sharp \to \mathsf{expr} \to \mathbb{T}^\sharp + \bot$ which returns the abstract taint corresponding to the evaluation of an expression. We use $\mathcal{T}(a, b) = b$ as tainting function, the companion of the erasure function $\mathcal{E}$.

$$\overline{\mathsf{taint}}^\sharp(\overline{\sigma}^\sharp, n) = \mathcal{L}^\sharp$$

$$\overline{\mathsf{taint}}^\sharp(\overline{\sigma}^\sharp, a) = \mathcal{L}^\sharp$$

$$\overline{\mathsf{taint}}^\sharp(\overline{\sigma}^\sharp, x) = \mathcal{T}(\overline{\sigma}^\sharp)(x)$$

$$\overline{\mathsf{taint}}^\sharp(\overline{\sigma}^\sharp, e_1 \oplus e_2) = \overline{\mathsf{taint}}^\sharp(\overline{\sigma}^\sharp, e_1) \sqcup^\sharp \overline{\mathsf{taint}}^\sharp(\overline{\sigma}^\sharp, e_2)$$

We now define the following abstract operators (i.e., transfer functions).

- $\overline{\mathsf{eval}}^\sharp(\overline{\sigma}^\sharp, e) = (\mathsf{eval}^\sharp(\mathcal{E}(\overline{\sigma}^\sharp), e), \overline{\mathsf{taint}}^\sharp(\overline{\sigma}^\sharp, e))$
- $\overline{\mathsf{assign}}^\sharp(\overline{\sigma}^\sharp, x, e) = (\mathsf{assign}^\sharp(\mathcal{E}(\overline{\sigma}^\sharp), x, e), \mathcal{T}(\overline{\sigma}^\sharp)[x \mapsto \overline{\mathsf{taint}}^\sharp(\overline{\sigma}^\sharp, e)])$
- $\overline{\mathsf{assert}}^\sharp(\overline{\sigma}^\sharp, e) = (\mathsf{assert}^\sharp(\mathcal{E}(\overline{\sigma}^\sharp), e), \mathcal{T}(\overline{\sigma}^\sharp))$

- $\overline{\mathsf{store}}^\sharp(\overline{\sigma}^\sharp, e_1, e_2) = (\mathsf{store}^\sharp(\mathcal{E}(\overline{\sigma}^\sharp), e_1, e_2), \mathcal{T}(\overline{\sigma}^\sharp)[l \mapsto \overline{\mathsf{taint}}^\sharp(\overline{\sigma}^\sharp, e_2)]_{l \in \mathsf{concretize}^\sharp(\mathsf{eval}^\sharp(\mathcal{E}(\overline{\sigma}^\sharp), e_1))})$
- $\overline{\mathsf{load}}^\sharp(\overline{\sigma}^\sharp, x, e) = (\mathsf{load}^\sharp(\mathcal{E}(\overline{\sigma}^\sharp), x, e), \mathcal{T}(\overline{\sigma}^\sharp)[x \mapsto \sqcup^\sharp_{l \in \mathsf{concretize}^\sharp(\mathsf{eval}^\sharp(\mathcal{E}(\overline{\sigma}^\sharp), e))} \mathcal{T}(\overline{\sigma}^\sharp)(l)])$

The definitions of $\overline{\mathsf{eval}}^\sharp$, $\overline{\mathsf{assign}}^\sharp$ and $\overline{\mathsf{assert}}^\sharp$ reuse the operators of $\mathsf{AbMem}(\mathbb{V}^\sharp, \mathbb{M}^\sharp)$ and modify slightly the tainting part. The definitions of $\overline{\mathsf{store}}^\sharp$ and $\overline{\mathsf{load}}^\sharp$ are more complex. In both cases, we need to use $\mathsf{eval}^\sharp$ to deduce all possible locations affected by the memory accesses and suitably update the tainting parts. For $\overline{\mathsf{store}}^\sharp(\overline{\sigma}^\sharp, e_1, e_2)$, all possible write locations given by the concretization of $\mathsf{eval}^\sharp(\mathcal{E}(\overline{\sigma}^\sharp), e_1)$ are updated, as for $\overline{\mathsf{load}}^\sharp(\overline{\sigma}^\sharp, x, e)$, we approximate the taints from all possible read locations given by the concretization of $\mathsf{eval}^\sharp(\mathcal{E}(\overline{\sigma}^\sharp), e)$. This concludes the definition of $\mathsf{AbMem}(\overline{\mathbb{V}}^\sharp, \overline{\mathbb{M}}^\sharp)$.

Finally, the abstract analysis $[\![p]\!](\sigma^\sharp, \tau^\sharp)$ of program p starting with tainted abstract environment $\overline{\sigma}^\sharp$ is defined in Figure 7. To analyze $(p_1; p_2)$, first p_1 is analyzed and then p_2 is analyzed using the environment given by the first analysis. Similarly, to analyze a statement $(\texttt{if } e \texttt{ then } p_1 \texttt{ else } p_2)$, p_1 is analyzed assuming that e is true and p_2 is analyzed assuming the opposite, $\sqcup^\sharp$ is then used to get an over-approximation of both results.

The loop $(\texttt{while } e \texttt{ do } p)$ is the trickiest part to analyze, as the analysis cannot just analyze one iteration of the loop body and then recursively analyze the loop again since this may never terminate. The analysis thus tries to find a loop invariant. The standard method in abstract interpretation is to compute a post-fixpoint of the function $\mathsf{iter}(e, p, \overline{\sigma}^\sharp_0, \cdot)$ as defined in Figure 7. It represents a loop invariant, the final result is thus the invariant where the test condition does not hold anymore. In order to compute the post-fixpoint, we use $\mathsf{pfp}(f)$ which computes a post-fixpoint of monotone function f by successively computing $\bot, f(\bot), f(f(\bot)), \ldots$, and forces convergence using a widening-narrowing operator on the $\mathbb{M}^\sharp$ part. Computing $\bot, f(\bot), f(f(\bot)), \ldots$ converges toward the least fixpoint (i.e., the most precise invariant) when the process terminates according to Kleene theorem, but this process is not guaranteed to end. The usual solution in abstract interpretation to ensure termination is to use a widening operator ∇ which overapproximates the limit of $\bot, f(\bot), f(f(\bot)), \ldots$ by instead computing the sequence $x_0 = \bot, x_{n+1} = x_n \nabla f(x_n)$ which converges in a finite number of step. However, the resulting post-fixpoint may be grossly imprecise, and a narrowing operator Δ can be used in the same way to improve precision. The taint part does not require convergence help because taints form a finite lattice.

3.5. Correctness of the Abstract Interpreter

In order to specify and prove the correctness of the analyzer, we follow the usual methodology in abstract interpretation and define a *collecting* semantics, aiming at facilitating the proof. The semantics still expresses the dynamic behavior of programs but takes a closer form to the analysis. It operates over properties of concrete environments, thus bridging the gap between concrete environments and abstract environments, which represent sets of concrete environments.

The collecting semantics aims at describing the resulting environments that can be reached given a specific instruction and a set of environments. The collecting semantics of a program p with a set of concrete environments Σ is written $[\![p]\!](\Sigma)$.

Similarly to the abstract interpreter, we define $\overline{\mathsf{Assign}}$, $\overline{\mathsf{Store}}$, $\overline{\mathsf{Load}}$, $\overline{\mathsf{Assert}}$. They will respectively serve as counterparts to $\overline{\mathsf{assign}}^\sharp$, $\overline{\mathsf{store}}^\sharp$, $\overline{\mathsf{load}}^\sharp$ and $\overline{\mathsf{assert}}^\sharp$. We first start with $\overline{\mathsf{Assign}}$:

$$\overline{\mathsf{Assign}}(\Sigma, x, e) = \{\overline{\sigma}[x \mapsto (v, t)] | \exists v \in \mathbb{V}, t \in \mathbb{T}, \overline{\sigma}(e) = (v, t) \wedge \overline{\sigma} \in \Sigma\}$$

$$[\![\texttt{skip}]\!]^\sharp(\overline{\sigma}^\sharp) = \overline{\sigma}^\sharp$$

$$[\![*e_1 \leftarrow e_2]\!]^\sharp(\overline{\sigma}^\sharp) = \overline{\mathsf{store}}^\sharp(\overline{\sigma}^\sharp, e_1, e_2)$$

$$[\![x \leftarrow *e]\!]^\sharp(\overline{\sigma}^\sharp) = \overline{\mathsf{load}}^\sharp(\overline{\sigma}^\sharp, x, e)$$

$$[\![x \leftarrow e]\!]^\sharp(\overline{\sigma}^\sharp) = \overline{\mathsf{assign}}^\sharp(\overline{\sigma}^\sharp, x, e)$$

$$[\![p_1; p_2]\!]^\sharp(\overline{\sigma}^\sharp) = [\![p_2]\!]^\sharp([\![p_1]\!]^\sharp(\overline{\sigma}^\sharp))$$

$$[\![\texttt{if } e \texttt{ then } p_1 \texttt{ else } p_2]\!]^\sharp(\overline{\sigma}^\sharp) = [\![p_1]\!]^\sharp(\overline{\mathsf{assert}}^\sharp(\overline{\sigma}^\sharp, e)) \sqcup^\sharp [\![p_2]\!]^\sharp(\overline{\mathsf{assert}}^\sharp(\overline{\sigma}^\sharp, \mathsf{not}\ e))$$

$$[\![\texttt{while } e \texttt{ do } p]\!]^\sharp(\overline{\sigma}_0^\sharp) = \overline{\mathsf{assert}}^\sharp(\mathsf{pfp}(\mathsf{iter}(e, p, \overline{\sigma}_0^\sharp, \cdot)), \mathsf{not}\ e)$$

$$\mathsf{iter}(e, p, \overline{\sigma}_0^\sharp, \overline{\sigma}^\sharp) = \overline{\sigma}_0^\sharp \sqcup^\sharp \overline{\mathsf{assert}}^\sharp([\![p]\!]^\sharp(\overline{\sigma}^\sharp), e)$$

Figure 7. Abstract execution of statements

Given a set of concrete environments Σ, $\overline{\mathsf{Assign}}(\Sigma, x, e)$ computes the set of all possible reachable environments from environments in Σ after executing $x \leftarrow e$ in the tainting semantics.

Next are $\overline{\mathsf{Store}}$ and $\overline{\mathsf{Load}}$:

$$\overline{\mathsf{Store}}(\Sigma, e_1, e_2) = \{\overline{\sigma}[l \mapsto (v,t)] | \exists l \in \mathbb{L}, v \in \mathbb{V}, t \in \mathbb{T}, \overline{\sigma}(e_1) = (l, \mathcal{L}) \wedge \overline{\sigma}(e_2) = (v,t) \wedge \overline{\sigma} \in \Sigma\}$$

$$\overline{\mathsf{Load}}(\Sigma, x, e) = \{\overline{\sigma}[x \mapsto (v,t)] | \exists l \in \mathbb{L}, v \in \mathbb{V}, t \in \mathbb{T}, \overline{\sigma}(e) = (l, \mathcal{L}) \wedge \overline{\sigma}(l) = (v,t) \wedge \overline{\sigma} \in \Sigma\}$$

Given a set of concrete environments Σ, $\overline{\mathsf{Store}}(\Sigma, e_1, e_2)$ (resp. $\overline{\mathsf{Load}}(\Sigma, x, e)$) computes the set of all possible reachable environments from environments in Σ after executing $*e_1 \leftarrow e_2$ (resp. $x \leftarrow *e$) in the tainting semantics.

$\overline{\mathsf{Assert}}$ removes the environments where e is not true:

$$\overline{\mathsf{Assert}}(\Sigma, e) = \{\overline{\sigma} \in \Sigma | \exists t, \overline{\sigma}(e) = (\mathsf{true}, t)\}$$

Finally, the collecting semantics is defined in Figure 8. Looking at the rules in Figure 7 and Figure 8, one can notice that the collecting semantics follows closely the shape of the abstract interpreter. The collecting semantics of assignment is defined using $\overline{\mathsf{Assign}}$, the counterpart of $\overline{\mathsf{assign}}^\sharp$. Similarly to the abstract interpreter, to evaluate conditional branchings, the first branch is evaluated assuming the condition is true using Assert and the second branch is evaluated assuming the opposite. The results are then merged to obtain all the possible states that can be reached.

We first start by proving that the collecting semantics is sound with regards to the tainting semantics.

Theorem 4. *For any programs p and environment $\overline{\sigma}$, $\langle \overline{\sigma}, p \rangle \rightsquigarrow^* \langle \overline{\sigma}', \texttt{skip} \rangle \implies \overline{\sigma}' \in [\![p]\!](\{\overline{\sigma}\})$.*

$$
\begin{aligned}
[\![\texttt{skip}]\!](\Sigma) &= \Sigma \\
[\![*e_1 \leftarrow e_2]\!](\Sigma) &= \overline{\mathsf{Store}}(\Sigma, e_1, e_2) \\
[\![x \leftarrow *e]\!](\Sigma) &= \overline{\mathsf{Load}}(\Sigma, x, e) \\
[\![x \leftarrow e]\!](\Sigma) &= \overline{\mathsf{Assign}}(\Sigma, x, e) \\
[\![p_1; p_2]\!](\Sigma) &= [\![p_2]\!]([\![p_1]\!](\Sigma)) \\
[\![\texttt{if } e \texttt{ then } p_1 \texttt{ else } p_2]\!](\Sigma) &= [\![p_1]\!](\overline{\mathsf{Assert}}(\Sigma, e)) \cup [\![p_2]\!](\overline{\mathsf{Assert}}(\Sigma, \mathsf{not}\ e)) \\
[\![\texttt{while } e \texttt{ do } p]\!](\Sigma) &= \overline{\mathsf{Assert}}(I, \mathsf{not}\ e)
\end{aligned}
$$

where I is the least fixpoint of $I == \Sigma \cup [\![p]\!](\mathsf{Assert}(I, e))$

Figure 8. Definition of the collecting semantics $[\![\cdot]\!](\cdot)$

Proof. This is a fairly standard proof in abstract interpretation. As the theorem statement does not directly fit well with induction, we first start by proving the following more general lemma:

$$\forall p, \overline{\sigma}, \overline{\sigma}', \Sigma, \overline{\sigma} \in \Sigma \implies \langle \overline{\sigma}, p \rangle \rightsquigarrow^* \langle \overline{\sigma}', \texttt{skip} \rangle \implies \overline{\sigma}' \in [\![p]\!](\Sigma)$$

The proof is by induction on p.

- If $p = \texttt{skip}$, it is trivially true.
- If $p = *e_1 \leftarrow e_2$ or $p = x \leftarrow *e$ or $p = x \leftarrow e$, it is true by definition of $\overline{\mathsf{Store}}$, $\overline{\mathsf{Load}}$, $\overline{\mathsf{Assign}}$ and by definition of the tainting semantics.
- If $p = p_1; p_2$, then there exists $\overline{\sigma}''$ such that $\langle \overline{\sigma}, p_1 \rangle \rightsquigarrow^* \langle \overline{\sigma}'', \texttt{skip} \rangle$ and $\langle \overline{\sigma}'', p_2 \rangle \rightsquigarrow^* \langle \overline{\sigma}', \texttt{skip} \rangle$. By induction hypothesis on the first execution, we obtain that $\overline{\sigma}'' \in [\![p_1]\!](\Sigma)$. Combining this with using the induction hypothesis on the second execution allows us to conclude that $\overline{\sigma}' \in [\![p_2]\!]([\![p_1]\!](\Sigma)) = [\![p_1; p_2]\!](\Sigma) = [\![p]\!](\Sigma)$.
- If $p = \texttt{if } e \texttt{ then } p_1 \texttt{ else } p_2$, then either $\overline{\sigma}(e) = \mathsf{true}$ and $\langle \overline{\sigma}, p_1 \rangle \rightsquigarrow^* \langle \overline{\sigma}', \texttt{skip} \rangle$ or $\overline{\sigma}(e) = \mathsf{false}$ and $\langle \overline{\sigma}, p_2 \rangle \rightsquigarrow^* \langle \overline{\sigma}', \texttt{skip} \rangle$. In the first case, $\overline{\sigma} \in \overline{\mathsf{Assert}}(\Sigma, e)$ and in the latter, $\overline{\sigma} \in \overline{\mathsf{Assert}}(\Sigma, \mathsf{not}\ e)$ which allows us to conclude in both cases by using the induction hypothesis.
- If $p = \texttt{while } e \texttt{ do } p$, then we know that $\overline{\sigma}'(e) = \mathsf{false}$. Furthermore, we remark that for all $\overline{\sigma}''$ such that $\langle \overline{\sigma}, \texttt{while } e \texttt{ do } p \rangle \rightsquigarrow^* \langle \overline{\sigma}'', \texttt{while } e \texttt{ do } p \rangle$, $\overline{\sigma}'' \in I$ by definition of I. Thus, $\overline{\sigma}' \in I$ and since $\overline{\sigma}'(e) = \mathsf{false}$, $\overline{\sigma}' \in \overline{\mathsf{Assert}}(I, \mathsf{not}\ e)$.

The lemma is thus proven, and the theorem is a direct consequence of it. □

The regular semantics also has a collecting semantics with the operators Assign, Store, Load, Assert and a corresponding soundness theorem that we will not detail. The operators are defined as follows:

$$
\begin{aligned}
&\mathsf{Assign}(\Sigma, x, e) = \{\sigma[x \mapsto v] | \exists v \in \mathbb{V}, \sigma(e) = v \wedge \sigma \in \Sigma\} \\
&\mathsf{Store}(\Sigma, e_1, e_2) = \{\sigma[l \mapsto v] | \exists l \in \mathbb{L}, v \in \mathbb{V}, \sigma(e_1) = l \wedge \sigma(e_2) = v \wedge \overline{\sigma} \in \Sigma\} \\
&\mathsf{Load}(\Sigma, x, e) = \{\sigma[x \mapsto v] | \exists l \in \mathbb{L}, v \in \mathbb{V}, \sigma(e) = l \wedge \sigma(l) = v \wedge \sigma \in \Sigma\} \\
&\mathsf{Assert}(\Sigma, e) = \{\sigma \in \Sigma | \sigma(e) = \mathsf{true}\}
\end{aligned}
$$

Finally, we also need to introduce the concept of concretization to state and prove the correctness of our abstract interpreter. We already introduced $\mathsf{concretize}^\sharp$ previously which is actually a concretization function. We will rename it $\gamma_{\mathbb{V}^\sharp}$ as γ is the usual name for a concretization function in abstract interpretation. We use $v \in \gamma_{\mathbb{V}^\sharp}(v^\sharp)$ to say that v is in the concretization of abstract value $v^\sharp$, which means that $v^\sharp$ represents a set of concrete values of which v is a member.

The abstract memory domain $\mathbb{M}^\sharp$ also provides a concretization function $\gamma_{\mathbb{M}^\sharp} : \mathbb{M}^\sharp \to \mathcal{P}(M)$ which is used to define the correctness of the $\mathsf{assign}^\sharp$, $\mathsf{store}^\sharp$, $\mathsf{load}^\sharp$ and $\mathsf{assert}^\sharp$ operators:

$$\mathsf{Assign}(\gamma_{\mathbb{M}^\sharp}(\sigma^\sharp), x, e) \subseteq \gamma_{\mathbb{M}^\sharp}(\mathsf{assign}^\sharp(\sigma^\sharp, e))$$

$$\mathsf{Store}(\gamma_{\mathbb{M}^\sharp}(\sigma^\sharp), e_1, e_2) \subseteq \gamma_{\mathbb{M}^\sharp}(\mathsf{store}^\sharp(\sigma^\sharp, e_1, e_2))$$

$$\mathsf{Load}(\gamma_{\mathbb{M}^\sharp}(\sigma^\sharp), x, e) \subseteq \gamma_{\mathbb{M}^\sharp}(\mathsf{load}^\sharp(\sigma^\sharp, x, e))$$

$$\mathsf{Assert}(\gamma_{\mathbb{M}^\sharp}(\sigma^\sharp), e) \subseteq \gamma_{\mathbb{M}^\sharp}(\mathsf{assert}^\sharp(\sigma^\sharp, e))$$

We now need to define $\gamma_{\mathbb{T}^\sharp} : \mathbb{T}^\sharp \to \mathcal{P}(\mathbb{T})$ and $\gamma_{\overline{\mathbb{M}}^\sharp} : \overline{\mathbb{M}}^\sharp \to \mathcal{P}(\overline{M})$.

The first one is simple, $\gamma_{\mathbb{T}^\sharp}(\mathcal{L}^\#) = \{\mathcal{L}\}$ and $\gamma_{\mathbb{T}^\sharp}(\mathcal{H}^\#) = \{\mathcal{L}, \mathcal{H}\}$. $\mathcal{L}^\#$ corresponds to value that we know are *necessarily* public data, while $\mathcal{H}^\#$ corresponds to value that we only know *may* depends on secrets.

Now, we define $\gamma_{\overline{\mathbb{M}}^\sharp}$:

$$\gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp) = \{\overline{\sigma} | \mathcal{E} \circ \overline{\sigma} \in \gamma_{\mathbb{M}^\sharp}(\mathcal{E}(\overline{\sigma}^\sharp)) \wedge \forall r, \mathcal{T}(\overline{\sigma}(r)) \in \gamma_{\mathbb{T}^\sharp}(\mathcal{T}(\overline{\sigma}^\sharp)(r))\}$$

This means that an environment $\overline{\sigma}$ is in the concretization of $\overline{\sigma}^\sharp$ if there exists $\sigma \in \gamma_{\mathbb{M}^\sharp}(\mathcal{E}(\overline{\sigma}^\sharp))$ such that $\mathcal{E} \circ \overline{\sigma} = \sigma$ and such that $\mathcal{T}(\overline{\sigma}(r)) \in \gamma_{\mathbb{T}^\sharp}(\mathcal{T}(\overline{\sigma}^\sharp)(r))$ for all location or variable r.

We now need to prove the correctness of the $\overline{\mathsf{assign}}^\sharp$, $\overline{\mathsf{store}}^\sharp$, $\overline{\mathsf{load}}^\sharp$ and $\overline{\mathsf{assert}}^\sharp$ operators:

$$\overline{\mathsf{Assign}}(\gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp), x, e) \subseteq \gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\mathsf{assign}}^\sharp(\overline{\sigma}^\sharp, x, e))$$

$$\overline{\mathsf{Store}}(\gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp), e_1, e_2) \subseteq \gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\mathsf{store}}^\sharp(\overline{\sigma}^\sharp, e_1, e_2))$$

$$\overline{\mathsf{Load}}(\gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp), x, e) \subseteq \gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\mathsf{load}}^\sharp(\overline{\sigma}^\sharp, x, e))$$

$$\overline{\mathsf{Assert}}(\gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp), e) \subseteq \gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\mathsf{assert}}^\sharp(\overline{\sigma}^\sharp, e))$$

Proof. We need to prove that for all $\overline{\sigma} \in \overline{\mathsf{Assign}}(\gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp), x, e)$, $\overline{\sigma} \in \gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\mathsf{assign}}^\sharp(\overline{\sigma}^\sharp, e))$. We first define $\mathcal{E}(\overline{\Sigma}) = \{\mathcal{E} \circ \overline{\sigma} | \overline{\sigma} \in \overline{\Sigma}\}$ for all $\overline{\Sigma} \in \mathcal{P}(\overline{M})$. We then notice that $\mathcal{E}(\overline{\mathsf{Assign}}(\gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp), x, e)) = \mathsf{Assign}(\gamma_{\mathbb{M}^\sharp}(\mathcal{E}(\overline{\sigma}^\sharp)), x, e))$ by definitions.

Then, by correctness of $\mathsf{assign}^\sharp$, we have that $\mathsf{Assign}(\gamma_{\mathbb{M}^\sharp}(\mathcal{E}(\overline{\sigma}^\sharp)), x, e)) \subseteq \gamma_{\mathbb{M}^\sharp}(\mathsf{assign}^\sharp(\mathcal{E}(\overline{\sigma}^\sharp), x, e))$. And by definition of $\overline{\mathsf{assign}}^\sharp$, we have that $\mathcal{E}(\overline{\mathsf{assign}}^\sharp(\overline{\sigma}, x, e)) = \mathsf{assign}^\sharp(\mathcal{E}(\overline{\sigma}^\sharp), x, e)$. Thus, $\gamma_{\mathbb{M}^\sharp}(\mathcal{E}(\overline{\mathsf{assign}}^\sharp(\overline{\sigma}, x, e))) = \gamma_{\mathbb{M}^\sharp}(\mathsf{assign}^\sharp(\mathcal{E}(\overline{\sigma}^\sharp), x, e))$ which implies that $\mathcal{E}(\overline{\mathsf{Assign}}(\gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp), x, e)) \subseteq \gamma_{\mathbb{M}^\sharp}(\mathcal{E}(\overline{\mathsf{assign}}^\sharp(\overline{\sigma}, x, e)))$ and therefore, there exists $\sigma \in \gamma_{\mathbb{M}^\sharp}(\mathcal{E}(\overline{\mathsf{assign}}^\sharp(\overline{\sigma}, x, e)))$ such that $\mathcal{E}(\overline{\sigma}) = \sigma$.

It is then left to prove that for all r, $\mathcal{T}(\overline{\sigma}(r)) \in \gamma_{\mathbb{T}^\sharp}(\mathcal{T}(\overline{\mathsf{assign}}^\sharp(\overline{\sigma}^\sharp, x, e))(r))$. By definition of $\overline{\mathsf{assign}}^\sharp$, $\mathcal{T}(\overline{\mathsf{assign}}^\sharp(\overline{\sigma}^\sharp, x, e)) = \mathcal{T}(\overline{\sigma}^\sharp)[x \mapsto \overline{\mathsf{taint}}^\sharp(\overline{\sigma}^\sharp, e)]$. By definition of $\overline{\mathsf{Assign}}$, we know that there exists $\overline{\sigma}_1 \in \gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp)$ such that $\overline{\sigma} = \overline{\sigma}_1[x \mapsto (v,t)]$ with $\overline{\sigma}_1(e) = (v,t)$.

The correctness of $\overline{\mathsf{taint}}^\sharp$ can easily be proven by induction on e:

$$\overline{\sigma} \in \gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp) \implies \mathcal{T}(\overline{\sigma}(e)) \in \gamma_{\mathbb{T}^\sharp}(\overline{\mathsf{taint}}^\sharp(\overline{\sigma}^\sharp, e))$$

By exploiting the lemma, the correctness of $\overline{\mathsf{assign}}^\sharp$ is thus proven. The correctness of the other operators is similarly proven. □

The following theorem which states the correctness of the abstract analyzer with regards to the collecting semantics can now be proven.

Theorem 5. *For all abstract environment $\overline{\sigma}^\sharp$ and program p,*

$$[\![p]\!](\gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp)) \subseteq \gamma_{\overline{\mathbb{M}}^\sharp}([\![p]\!]^\sharp(\overline{\sigma}^\sharp))$$

Proof. We first remark that $[\![p]\!]$ is a monotone function, i.e. $\Sigma_1 \subseteq \Sigma_2 \implies [\![p]\!](\Sigma_1) \subseteq [\![p]\!](\Sigma_2)$. The proof is by induction on p. The theorem is also proven by induction on p. We have that:

- if $p = \texttt{skip}$, it is trivially true;
- if $p = *e_1 \leftarrow e_2$ or $p = x \leftarrow e$ or $p = x \leftarrow *e$, it is a direct consequence of the correctness of the corresponding operators;
- if $p = p1; p2$, we have $[\![p_1]\!](\gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp)) \subseteq \gamma_{\overline{\mathbb{M}}^\sharp}([\![p_1]\!]^\sharp(\overline{\sigma}^\sharp))$ by induction hypothesis on p_1 and $[\![p_2]\!](\gamma_{\overline{\mathbb{M}}^\sharp}([\![p_1]\!]^\sharp(\overline{\sigma}^\sharp))) \subseteq \gamma_{\overline{\mathbb{M}}^\sharp}([\![p_2]\!]^\sharp([\![p_1]\!]^\sharp(\overline{\sigma}^\sharp)))$ on p_2. And by monotony of $[\![p_2]\!]$, we have $[\![p_1; p_2]\!](\gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp)) = [\![p_2]\!]([\![p_1]\!](\gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp))) \subseteq [\![p_2]\!](\gamma_{\overline{\mathbb{M}}^\sharp}([\![p_1]\!]^\sharp(\overline{\sigma}^\sharp))) \subseteq \gamma_{\overline{\mathbb{M}}^\sharp}([\![p_2]\!]^\sharp([\![p_1]\!]^\sharp(\overline{\sigma}^\sharp))) = \gamma_{\overline{\mathbb{M}}^\sharp}([\![p_1; p_2]\!]^\sharp(\overline{\sigma}^\sharp)$ which is what we needed to prove;
- if $p = \texttt{if } e \texttt{ then } p_1 \texttt{ else } p_2$, it is a consequence of the correctness of $\overline{\mathsf{assert}}^\sharp$;
- if $p = \texttt{while } e \texttt{ do } p$, it is a consequence of the correctness of pfp with regard with the invariant, and the correctness of $\overline{\mathsf{assert}}^\sharp$.

The theorem is thus proven. □

This theorem intuitively means that the abstract analyzer is correct with regards to the collecting semantics since if $\gamma_{\overline{\mathbb{M}}^\sharp}([\![p]\!]^\sharp(\overline{\sigma}^\sharp))$ is empty, $[\![p]\!](\gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp))$ must necessarily be empty too, and thus the execution is stuck with regards to the collecting semantics.

Finally, combining Theorems 4 and 5, the following correctness theorem is a direct consequence:

Theorem 6. *For all program p, environment $\overline{\sigma}$ and abstract environment $\overline{\sigma}^\sharp$ such that $\overline{\sigma} \in \gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp)$, if we have the execution $\langle \overline{\sigma}, p\rangle \rightsquigarrow^* \langle \overline{\sigma}', \texttt{skip}\rangle$, then we also have $\overline{\sigma}' \in \gamma_{\overline{\mathbb{M}}^\sharp}([\![p]\!]^\sharp(\overline{\sigma}^\sharp))$.*

This is the main theorem of correctness of the abstract interpreter. It ensures that we compute correct over-approximations of reachable states in the tainting semantics. We can then safely perform abstract tests on the program to check that no tainting state may reach a stuck configuration. By that, we mean that

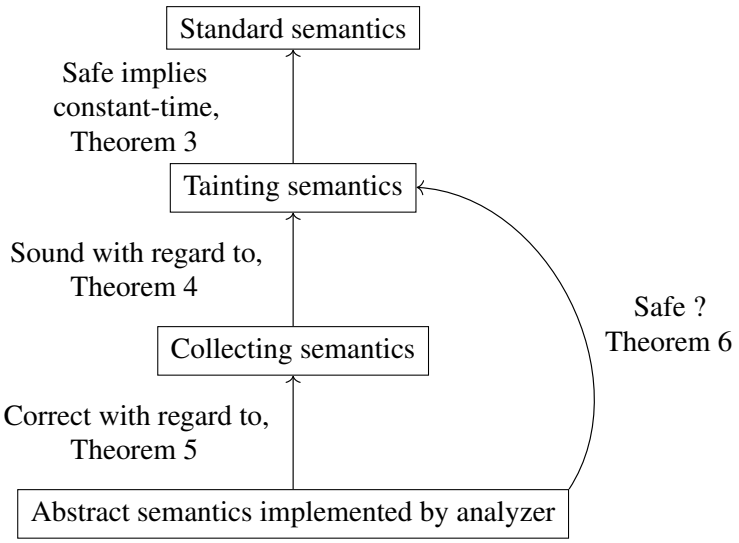

Figure 9. Diagram relating the different semantics

the analyzer may fail or raise alarms during the analysis. For instance, when analyzing `if (e) ...`, it may raise an alarm to say that `e` may potentially depend on a secret (i.e., it has a high taint) at this program point. Hence, we can conclude that if no alarm is raised, then the program is safe with regard to the tainting semantics and is thus constant-time.

Finally, Figure 9 summarizes the relationships between the different semantics and the theorems that links them.

4. Implementation and Experiments

Following the methodology presented in Section 3, we have implemented a prototype leveraging the Verasco static analyzer. We have been able to evaluate our prototype by verifying multiple actual C code constant-time algorithms taken from different cryptographic libraries such as NaCl [5], mbedTLS [6], Open Quantum Safe [21] and `curve25519-donna` [22]. The implementation is available at the following address http://www.irisa.fr/celtique/ext/esorics17/.

In order to use our tool, the user simply has to indicate which variables are to be considered as secrets and the prototype will either raise alarms indicating where secrets may leak, or indicate that the input program is constant time. The user can either indicate a whole global variable to be considered as secret at the start of the program, or use the `verasco_any_int_secret` built-in function to produce a random signed integer to be considered as secret.

The While language we presented has a few differences with the C#minor language of CompCert that we analyze using Verasco. First, C#minor allows more constructs such as `switch` and does not use `while` loops, but infinite loops that must be exited using a `break` statement. Secondly, C#minor expressions can contain memory reads whereas our While language models a memory load as a statement. However, this is only a slight difference as C#minor programs such as $x = *y + *z$ are already transformed into $x1 = *y; x2 = *z; x = x1 + x2$ by Verasco in order to improve the precision of the analysis.

```
1  int main(void) {
2    int t[4] = { verasco_any_int(), verasco_any_int_secret(),
3         verasco_any_int(), verasco_any_int_secret() };
4    for (int i = 0; i < 4; i++)
5       if (i%2 == 0) {  // First if condition
6         if (t[i])  t[i] = 0; } // Second if condition
7    return 0; }
```

Figure 10. An example program that is analyzed as constant time

4.1. Memory Separation

By leveraging Verasco, the prototype has no problem handling difficult problems such as memory separation. For example, the small example of Figure 10 is easily proven as constant time. In this program, an array `t` is initialized with random values, such that the values in odd offsets are considered as secrets, contrary to values in even offsets. So, the analyzer needs to be precise enough to distinguish between the array cells and to take into account pointer arithmetic. The potential leak happens on line 6. However, the condition on line 5 constrains `i%2 == 0` to be true, and thus `i` must be even on line 6, so `t[i]` does not contain a secret. A naive analyzer would taint the whole array as secret and would thus not be able to prove the program constant-time, however our prototype has no problem to prove it.

Interestingly, an illustration of the problem can be found in real-world programs. For example, the NaCl implementation of SHA-256 is not handled by [20] due to this. Indeed, in this program, the hashing function uses the following C `struct` as an internal state that contains both secret and public values during execution.

```
typedef struct crypto_hash_sha256_state {
  uint32_t      state[8];
  uint32_t      count[2];
  unsigned char buf[64];          } crypto_hash_sha256_state;
```

The hashing function is defined as follows.

```
int crypto_hash(unsigned char *out, const unsigned char *in,
                unsigned long long inlen)
{
    crypto_hash_sha256_state state;
    crypto_hash_sha256_init(&state);
    crypto_hash_sha256_update(&state, in, inlen);
    crypto_hash_sha256_final(&state, out);
    return 0;
}
```

It first starts by initializing the internal state with some constant value and then updates it using the input value `in` which is considered secret as it can be a password that an user is trying to hash. Both fields `state` and `buf` may contain secret dependent values as a result of the update. Last, `crypto_hash_sha256_final` contains a conditional branching that depends on the `count` field of the internal state: `if ((state->count[1] += bitlen[1]) < bitlen[1])`. However, the

Example	Size	Loc	Time	Result
aes	1171	1399	41.39	
curve25519-donna	1210	608	586.20	✓
des	229	436	2.28	
rlwe_sample	145	1142	30.76	✓
salsa20	341	652	5.34	✓
sha3	531	251	57.62	✓
snow	871	460	4.37	✓
tea	121	109	3.47	✓
bear_aes_ct	803	766	1.97	✓
bear_des_ct	454	560	2.54	✓
bear_sha1	243	197	2.45	✓
bear_sha256	259	329	2.83	✓
nacl_chacha20	384	307	0.34	✓
nacl_sha256	368	287	1.85	✓
mbedtls_sha1	544	354	0.33	✓
mbedtls_sha256	346	346	0.62	✓
mbedtls_sha512	310	399	0.58	✓
mee-cbc	1959	939	933.37	✓

Table 1
Verification of cryptographic primitives

whole internal state `struct` is allocated as a single memory block at low level (i.e., LLVM) and [20] does not manage to prove the memory separation and cannot thus ensure that the hashing function is secure.

4.2. Cryptographic Algorithms

We report in Table 1 our results on a set of cryptographic algorithms. All executions times reported were obtained on a 3.1GHz Intel i7 with 16GB of RAM. Sizes are reported in terms of numbers of C#minor statements (i.e., close to C statements), lines of code are measured with `cloc` and execution times are reported in seconds. The last column indicates whether the corresponding program has been managed to be secure by the analysis.

The first block of lines gathers test cases for the implementations of a representative set of cryptographic primitives including TEA [23], an implementation of sampling in a discrete Gaussian distribution by Bos et al. [21] (`rlwe_sample`) taken from the Open Quantum Safe library [7], an implementation of elliptic curve arithmetic operations over Curve25519 [24] by Langley [22](`curve25519-donna`), and various primitives such as AES, DES, etc. The second block reports on implementations from the BearSSL library [25]. The third block reports on different implementations from the NaCl library [5]. The fourth block reports on implementations from the mbedTLS [6] library. Finally, the last result corresponds to an implementation of MAC-then-Encode-then-CBC-Encrypt (MEE-CBC).

All the analyses are done using the interval abstract domain to track numerical values as it is sufficient in our case and is the fastest. Prior to analysis, the programs are slightly transformed using the `funload` option of Verasco which lifts loads out of expressions. For instance, `x = *(p + 3) + 4` is transformed into `y = *(p + 3); x = y + 4` where `y` is a fresh variable. This allows the analysis to be more precise. Furthermore, Verasco is not able to compute a precise enough loop invariant for some

of the programs, we thus indicate to Verasco to unroll these loops. Some of the timings differ with the results reported in the conference version due to a bug in the modification made to Verasco which caused it to silently fail. Unfortunately, `nacl_sha512` was erroneously reported as verified in the conference version.

All these examples are proven constant time, except for AES and DES which both make use of look up tables. Our prototype rightfully reports memory accesses depending on secrets, so these two programs are not constant time. Similarly to [20], `rlwe_sample` is only proven constant time, provided that the core random generator is also constant time, thus showing that it is the only possible source of leakage.

The last example `mee-cbc` is a full implementation of the MEE-CBC construction using low-level primitives taken from the NaCl library. Our prototype is able to verify the constant-time property of this example, showing that it scales to large code bases (939 loc).

Our prototype is able to verify a similar amount of programs than [20], except for a constant-time fixed point operations library named `libfixedtimefixedpoint` [26] which unfortunately does not use standard C and is not handled by CompCert. The library uses extensively a GNU extension known as statement-expressions and would require heavy rewriting to be accepted by our tool.

On the other hand, our tool shows its agility with memory separation on the program SHA-256 that was out of reach for [20] and its restricted alias management. In terms of analysis time, our tool behaves similarly to [20]. On a similar experiment platform, we observe a speedup between 0.1 and 10. This is very encouraging for our tool whose efficiency is still in an upgradeable stage, compared to the tool of [20] that relies on decades of implementation efforts for the LLVM optimizer and the Boogie verifier.

5. Related Work

This paper deals with static program verification for **information-flow tracking** [12]. Different formal techniques have been used in this area. The type-based approach [27] provides an elegant modular verification approach but requires program annotations, especially for each program function. Because a same function can be called in very different security contexts, providing an expressive annotation language is challenging and annotating programs is a difficult task. This approach has been mainly proposed for programming languages with strong typing guarantees such as Java [27] and ML [28]. The deductive approach [29] is based on more expressive logics than type systems and then allows to express subtle program invariants. On the other hand, the loop invariant annotation effort requires strong formal method expertise and is very much time consuming.

The static analysis approach only requires minimal annotation (the input taints) and then tries to infer all the remaining invariants in the restricted analysis logic. This approach has been followed to track efficiently implicit flows using program dependence graphs [30, 31]. We also follow a static approach but our backbone technique is an advanced value analysis for C, that we use to infer fine-grained memory separation properties and finely track taints in an unfolded call graph of the program. Building a program dependence graph for memory is a well-known challenge and scaling this approach to a Verasco (or Astrée) memory analysis is left for further work.

This paper deals however with a restricted notion of information flow: **constant-time security**. Here, implicit flow tracking is simplified since we must reject[1] control-flow branching that depends on secret inputs. Our abstract interpretation approach proposes to companion a taint analysis with a powerful value

[1] We could accept some of them if we were able to prove that all branches provide a similar timing behavior.

analysis. The tool tis-ct [32] uses a similar approach but based on the Frama-C value analysis, instead of Verasco (and its Astrée architecture). The tool is developed by the TrustInSoft company and not associated with any scientific publication. It has been used to analyze OpenSSL. Frama-C and Verasco value analysis are based on different abstract interpretation techniques and thus the tainting power of tis-ct and our tool will differ. As an example of difference, Verasco provides relational abstraction of memory contents while tis-ct is restricted to non-relational analysis (like intervals). CacheAudit [33] is also based on abstract interpretation but analyzes cache leakage at binary level. Analyzing program at this low level tempers the inference capabilities for memory separation, because the memory is seen as a single memory block. Verasco benefits from a source-level view where each function has its own region for managing local variables.

In a previous work of the second author [34], C programs were compiled by CompCert to an abstraction of assembly before being analyzed. A simple data-flow analysis was then performed, flow insensitive for every memory block except the memory stack, and constant-time security was verified. The precision of this approach required to fully inline the program before analysis. It means that every function call was replaced by its function body until no more function call remained. This has serious impact on the efficiency of the analysis and a program like `curve25519-donna` was out of reach. The treatment of memory stack was also very limited since no value analysis was available at this level of program representation. There was no way to finely taint an array content if this array laid in the stack (which occurs when C arrays are declared as local variables). Hence, numerous manual program rewritings were required before analysis. Our current approach releases these restrictions but requires more trust on the compiler (see our discussion in the conclusion).

A very complete treatment of constant-time security has been recently proposed by the ct-verif tool [20]. Its verification is based on a reduction of constant-time security of a program P to safety of a *product program* Q that simulates two parallel executions of P. The tool is based on the LLVM compiler and operates at the LLVM bytecode level, after LLVM optimizations and alias analyses. Once obtained, the bytecode program is transformed into a product program which, in turn, is verified by the Boogie verifier [35] and its traditional SMT tool suite. In Section 4, we made a direct experimental comparison with this tool. We list here the main design differences between this work and ours.

First we do not perform the analysis at a similar program representation. LLVM bytecode is interesting because one can develop analyses that benefit from the rich collection of tools provided by the LLVM platform. For example, [20] benefits from LLVM data-structure analysis [36] to partition memory objects into disjoint regions. Still, compiler alias analyses are voluntarily limited because compilers translate programs in almost linear time. Verasco (and its ancestor Astrée) follows a more ambitious approach and tracks very finely the content of the memory. Using Verasco requires a different tool design but opens the door for more verified programs, as for example the SHA-256 example. Second, we target a more restricted notion of constant-time security than [20] which relaxes the property with a so-called notion of *publicly observable outputs*. The extension is out of scope of our current approach but seems promising for specific programs. Only one program in our current experiment is affected by this limitation. At last, we embed our tool in a more foundational semantic framework. Verasco and CompCert are formally verified. It leaves the door open for a fully verified constant-time analyzer, while a fully verified ct-verif tool would require to prove SMT solvers, Boogie verifier and LLVM. The Vellvm [37] is a first attempt in the direction of verifying the LLVM platform, but it is currently restricted to a core subset (essentially the SSA generation) of the LLVM passes, and suffers from time-performance limitations.

Constant-time security is part of the larger field of **high-assurance cryptography** [38] which is a fertile area that has spawned many recent projects. There are two broad categories of methods of ensuring

high assurance: either it is formally verified using a proof assistant such as Coq or F*, or it is verified using automatic tools such as Boogie. Each method has its own drawbacks: the former usually needs a highly experienced user to be accomplished while the latter is automatic but needs to trust in unverified and non-trivial tools such as SMT solvers. Our work places itself in the first category.

Vale [39] is a tool for producing verified cryptographic assembly code. Users write code in the Vale language which is similar to assembly, and then add a functional specification of the code in Dafny [40], an automatic program verifier. The tool then automatically verifies that the code complies with the specification by using SMT solvers such as Z3 [41]. The authors also implemented a verified analyzer to ensure absence of timing and cache based side channels. However, like [20], their analyzer does not handle memory separation problems well and has to rely on handwritten annotation from the user. For instance, they also cannot automatically handle the same SHA256 example as [20].

HACL* [42] is a formally verified C cryptographic library. Similarly to [39], the library was created by first writing cryptographic code in the proof assistant F* [43]. The code is then verified for functional correctness, memory safety and freedom of timing side channels. However, unlike [39], proofs are entirely manual and thus need an experienced user. While their work tackles functional correctness and constant-time security of cryptographic programs, an experienced user is needed, whereas ours is automatic but only deals with constant-time security. Furthermore, as F* is a high-level language, it is more difficult to make sure that these properties are preserved during compilation, while our work focuses on C#minor which is a language closer to assembly than F*.

Jasmin [44] is a formally-verified compiler from the Jasmin language down to assembly. The Jasmin language is a small low-level language similar to Bernstein's qhasm that also supports function calls and high-level control-flow constructs such as loops. The authors have implemented a sound embedding of Jasmin into Dafny and users can thus automatically prove memory safety and constant-time security of their Jasmin programs using SMT solvers. Constant-time security is proven using product programs similarly to the ct-verif tool [20]. However, they do not mention if they suffer from the same memory separation issues.

FaCT [45] proposes a domain-specific language (DSL) to replace C as it is very prone to errors that can enable side channels. Their DSL can be basically seen as C enhanced with new annotations for expressing security levels such as which inputs can be considered secret or public. The language contains also new instructions that directly map to useful hardware instructions such as add-with-carry that are rarely produced by general purpose compilers. They use the Z3 SMT solver to prove memory safety of code written in this new language. Furthermore, as secret and public annotations are built in the language, they can adjust the compiler in order to take advantage of constant-time aware optimizations. Finally, as the tool is built upon LLVM, they can use the ct-verif tool [20] to verify that the generated code is secure, but must thus suffer the same limitations.

Fiat-crypto [46] is a formally verified compiler specifically optimized for generating efficient elliptic-curve code used in cryptography. However, the proven properties are only concerned with functional correctness. The compilation results in straightline code and they thus do not have to worry about secret dependent branching, but only about secret dependent memory accesses. The resulting code is thus not entirely proven constant-time.

In a series of publications [47–49], the authors leverage the Verified Software Toolchain and CompCert to prove the functional correctness of an ASM implementation of SHA-256 and an implementation of HMAC with SHA-256, as well as functional correctness and cryptographic security of an implementation of HMAC-DBRG. However, they do not prove anything about side-channels resistance.

Other approaches rely on dynamic analysis (e.g. [50] that extends of Valgrind in order to check constant-address security) or on statistical analysis of execution timing [51]. These approaches are not sound, contrary to our approach.

6. Conclusion

In this paper, we presented a methodology to ensure that a software implementation is constant time. Our methodology is based on advanced abstract interpretation techniques and scales on commonly used cryptographic libraries. Our implementation sits in a rich foundational semantic framework, Verasco and CompCert, which give strong semantic guarantees. The analysis is performed at source level and can hence give useful feedback to the programmer that needs to understand why his program is not constant time.

There are multiple possible directions for future work. The first one concerns semantic soundness. By inspecting CompCert transformation passes, we conjecture that they preserve the constant-time property of source programs that have been successfully analyzed. Informally, no conditional branching is added during compilation. Furthermore, memory accesses can only be added due to spilling during register allocation; however, these spilled variables are allocated at constant offsets on the stack and cannot thus depend on secrets. We left as further work a formal proof of this conjecture.

A second direction concerns expressiveness. In order to verify more relaxed properties, we could try to mix the program-product approach of [20] with the Verasco analysis. The current loop invariant inference and analysis of [20] are rather restricted. Using advanced alias analysis and relational numeric analysis could strengthen the program-product approach, if it was performed at the same representation level as Verasco.

Another possible direction is to also verify that multiplications and divisions are not secret dependent as on some platforms, their execution times may differ depending on the data they operate on.

Acknowledgements

This work is supported by a European Research Council (ERC) Consolidator Grant for the project VESTA, funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 772568).

References

[1] O. Aciiçmez, Ç.K. Koç and J.-P. Seifert, On the Power of Simple Branch Prediction Analysis, in: *ACM Symposium on Information, Computer and Communications Security*, ACM, 2007, pp. 312–320.
[2] N.J.A. Fardan and K.G. Paterson, Lucky thirteen: Breaking the TLS and DTLS record protocols, in: *Symp. on Security and Privacy (SP 2013)*, IEEE Computer Society, 2013, pp. 526–540.
[3] P. Kocher, Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems, in: *Advances in Cryptology — CRYPTO '96*, Springer, ed., LNCS, Vol. 1109, 1996, pp. 104–113.
[4] B. Canvel, A. Hiltgen, S. Vaudenay and M. Vuagnoux, Password Interception in a SSL/TLS Channel, in: *CRYPTO, volume 2729 of LNCS*, Springer, 2003, pp. 583–599.
[5] D.J. Bernstein, T. Lange and P. Schwabe, The security impact of a new cryptographic library, in: *International Conference on Cryptology and Information Security in Latin America*, Springer, 2012, pp. 159–176.
[6] mbed TLS (formerly known as PolarSSL), https://tls.mbed.org/.
[7] Open Quantum Safe, https://openquantumsafe.org/.

[8] M. Zhao and G.E. Suh, FPGA-Based Remote Power Side-Channel Attacks, in: *2018 IEEE Symposium on Security and Privacy (SP)*, Vol. 00, pp. 839–854. ISSN 2375-1207. doi:10.1109/SP.2018.00049.
[9] X. Leroy, Formal verification of a realistic compiler, *Communications of the ACM* **52**(7) (2009), 107–115.
[10] D.E. Denning, A Lattice Model of Secure Information Flow, *Commun. ACM* **19**(5) (1976), 236–243.
[11] D. Hedin and A. Sabelfeld, A Perspective on Information-Flow Control, in: *Software Safety and Security - Tools for Analysis and Verification*, IOS Press, 2012, pp. 319–347.
[12] A. Sabelfeld and A.C. Myers, Language-based information-flow security, *IEEE Journal on Selected Areas in Communications* **21**(1) (2003), 5–19.
[13] P. Cousot and R. Cousot, Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, in: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages,*, ACM, 1977, pp. 238–252.
[14] A. Miné, The octagon abstract domain, *Higher-Order and Symbolic Computation* **19**(1) (2006), 31–100.
[15] J. Feret, Static Analysis of Digital Filters, in: *European Symposium on Programming (ESOP'04)*, LNCS, Springer, 2004.
[16] A. Miné, Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics, in: *Proc. of The ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (*LCTES'06*), ACM, 2006, pp. 54–63.
[17] S. Blazy, V. Laporte and D. Pichardie, An Abstract Memory Functor for Verified C Static Analyzers, in: *Proc. of the 21st ACM SIGPLAN Int. Conference on Functional Programming (ICFP 2016)*, ACM, 2016.
[18] J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy and D. Pichardie, A Formally-Verified C Static Analyzer, in: *Proc. of the 42th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015*, ACM, 2015, pp. 247–259.
[19] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux and X. Rival, A static analyzer for large safety-critical software, in: *Programming Language Design and Implementation*, ACM, 2003, pp. 196–207.
[20] J.B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir and M. Emmi, Verifying Constant-Time Implementations, in: *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, 2016, pp. 53–70.
[21] J.W. Bos, C. Costello, M. Naehrig and D. Stebila, Post-Quantum Key Exchange for the TLS Protocol from the Ring Learning with Errors Problem, in: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, 2015, pp. 553–570.
[22] donna, https://code.google.com/archive/p/curve25519-donna.
[23] D.J. Wheeler and R.M. Needham, *TEA, a tiny encryption algorithm*, in: *Fast Software Encryption: Second International Workshop Leuven, Belgium, December 14–16, 1994 Proceedings*, B. Preneel, ed., Springer Berlin Heidelberg, Berlin, Heidelberg, 1995, pp. 363–366. ISBN 978-3-540-47809-6.
[24] D.J. Bernstein, *Curve25519: New Diffie-Hellman Speed Records*, in: *Public Key Cryptography - PKC 2006: 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24-26, 2006. Proceedings*, M. Yung, Y. Dodis, A. Kiayias and T. Malkin, eds, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 207–228. ISBN 978-3-540-33852-9.
[25] BearSSL, https://www.bearssl.org/.
[26] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner and H. Shacham, On Subnormal Floating Point and Abnormal Timing, in: *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, IEEE Computer Society, Washington, DC, USA, 2015, pp. 623–639. ISBN 978-1-4673-6949-7.
[27] A.C. Myers, JFlow: Practical Mostly-Static Information Flow Control, in: *Proc. of POPL'99*, 1999, pp. 228–241.
[28] F. Pottier and V. Simonet, Information flow inference for ML, *ACM Trans. Program. Lang. Syst.* **25**(1) (2003), 117–158.
[29] Ádám Darvas, R. Hähnle and D. Sands, A Theorem Proving Approach to Analysis of Secure Information Flow, in: *Proc. 2nd International Conference on Security in Pervasive Computing*, D. Hutter and M. Ullmann, eds, LNCS, Vol. 3450, Springer-Verlag, 2005, pp. 193–209.
[30] C. Hammer and G. Snelting, Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs, *Int. J. Inf. Sec.* **8**(6) (2009), 399–422.
[31] B. Rodrigues, F.M. Quintão Pereira and D.F. Aranha, Sparse Representation of Implicit Flows with Applications to Side-channel Detection, in: *Compiler Construction*, ACM, 2016, pp. 110–120.
[32] TIS-CT, http://trust-in-soft.com/tis-ct/.
[33] G. Doychev, D. Feld, B. Köpf, L. Mauborgne and J. Reineke, CacheAudit: A Tool for the Static Analysis of Cache Side Channels, in: *USENIX Conference on Security*, USENIX Association, Berkeley, CA, USA, 2013, pp. 431–446.
[34] G. Barthe, G. Betarte, J.D. Campo, C.D. Luna and D. Pichardie, System-level Non-interference for Constant-time Cryptography, in: *ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 1267–1279.
[35] M. Barnett, B.E. Chang, R. DeLine, B. Jacobs and K.R.M. Leino, Boogie: A Modular Reusable Verifier for Object-Oriented Programs, in: *Proc. of FMCO 2005*, 2005, pp. 364–387.
[36] C. Lattner, A. Lenharth and V.S. Adve, Making context-sensitive points-to analysis with heap cloning practical for the real world, in: *Proc. of PLDI'07*, 2007, pp. 278–289.

[37] J. Zhao, S. Zdancewic, S. Nagarakatte and M. Martin, Formalizing the LLVM Intermediate Representation for Verified Program Transformation, in: *POPL'12*, ACM, New York, NY, USA, 2012, pp. 427–440.
[38] G. Barthe, High-assurance cryptography: Cryptographic software we can trust, *IEEE Security & Privacy* **13**(5) (2015), 86–89.
[39] B. Bond, C. Hawblitzel, M. Kapritsos, K.R.M. Leino, J.R. Lorch, B. Parno, A. Rane, S. Setty and L. Thompson, Vale: Verifying high-performance cryptographic assembly code, in: *Proceedings of the USENIX Security Symposium*, 2017.
[40] K.R.M. Leino, Dafny: An automatic program verifier for functional correctness, in: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, Springer, 2010, pp. 348–370.
[41] L. De Moura and N. Bjørner, Z3: An efficient SMT solver, in: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 337–340.
[42] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko and B. Beurdouche, HACL*: A Verified Modern Cryptographic Library, in: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, ACM, New York, NY, USA, 2017, pp. 1789–1806. ISBN 978-1-4503-4946-8. doi:10.1145/3133956.3134043.
[43] N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan and J. Yang, Secure distributed programming with value-dependent types, in: *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming*, M.M.T. Chakravarty, Z. Hu and O. Danvy, eds, ACM, 2011, pp. 266–278. ISBN 978-1-4503-0865-6. doi:10.1145/2034773.2034811. https://www.microsoft.com/en-us/research/publication/secure-distributed-programming-with-value-dependent-types/.
[44] J. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt and P.-Y. Strub, Jasmin: High-Assurance and High-Speed Cryptography, in: *CCS 2017-Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.
[45] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala and D. Stefan, FaCT: A Flexible, Constant-Time Programming Language, in: *Cybersecurity Development (SecDev), 2017 IEEE*, IEEE, 2017, pp. 69–76.
[46] A. Erbsen, J. Philipoom, J. Gross, R. Sloan and A. Chlipala, Simple High-Level Code For Cryptographic Arithmetic – With Proofs, Without Compromises, in: *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.
[47] A.W. Appel, Verification of a cryptographic primitive: SHA-256, *ACM Transactions on Programming Languages and Systems (TOPLAS)* **37**(2) (2015), 7.
[48] L. Beringer, A. Petcher, Q.Y. Katherine and A.W. Appel, Verified Correctness and Security of OpenSSL HMAC., in: *USENIX Security Symposium*, 2015, pp. 207–221.
[49] K.Q. Ye, M. Green, N. Sanguansin, L. Beringer, A. Petcher and A.W. Appel, Verified Correctness and Security of mbedTLS HMAC-DRBG, in: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, ACM, New York, NY, USA, 2017, pp. 2007–2020. ISBN 978-1-4503-4946-8. doi:10.1145/3133956.3133974.
[50] ctgrind, https://github.com/agl/ctgrind.
[51] O. Reparaz, J. Balasch and I. Verbauwhede, Dude, is my code constant time, in: *Proc. of DATE 2017*, 2017.